Command and Control Toolkit™

Administrator's Manual

2.0

Command and Control Toolkit™

Administrator's Manual

September 2002

COMMAND AND CONTROL TECHNOLOGIES
CORPORATION

Contributors

Written by Eric Sorton

Edited by Kevin Brown and Vicki Gardner

Production by Patti Schill and Amy Banta

Engineering contributions by Greg Hupf, Frank Noble, and John Ward

Corporate Headquarters

Command and Control Technologies Corp.
1425 Chaffee Drive, Suite 1
Titusville, FL 32780 USA
Tel: 321.264.1193
Fax: 321.383.5096
Web: http://www.cctcorp.com

Technical Support

Tel: 321.264.1193
Email: support@cctcorp.com
Web: http://www.cctcorp.com/support

CCTK Administrator’s Manual

MNL-CCTK-Admin-120302.doc

August 24, 2001

TABLE OF CONTENTS

1 INTRODUCTION

The Command and Control Toolkit™ product is a data acquisition and commanding software package that can be tailored to a wide range of command and control applications. The product, also known as CCTK, provides a customizable client/server environment designed to meet the performance requirements of mission critical command and control operations. The CCTK client/server environment can be easily expanded from a single display console to any number of networked display consoles.

The CCTK documentation set provides a comprehensive description of how to install, configure, and use the CCTK software. This documentation, together with the reference documents listed in each manual, contains the information you will need to apply CCTK and any optional modules you have purchased to your command and control operation.

1.1 Manual Organization

This document describes configuration and administrative tasks associated with the Command and Control Toolkit™ (CCTK). It is intended to guide users familiar with the operation of CCTK through the steps necessary to configure the system. This manual contains important information for project administrators as well as power users of CCTK. The manual is written for users with a basic knowledge of the UNIX command line interface and a basic knowledge of XML. The concepts and terminology discussed in this document are defined in the *CCTK User's Manual*.

The first section after the introduction presents the user environment. Before configuring or executing CCTK, it is necessary to have a correctly configured environment. Section 2 shows how to configure the environment to execute CCTK.

Section 3 shows how to configure a CCTK project. This section covers the project configuration file, archive configuration, the configuration database, as well as several other minor topics. Reference this section when you need information about any of the configuration files associated with CCTK.

The next section discusses project execution. Methods for starting and stopping a project are introduced. The ability to start and stop applications is discussed as well as communicating with the project from a CCTK client. The artifacts created during project execution are introduced and explained. Finally, a set of utilities is presented that are useful during project operation.

Section 5 discusses operations that occur after a project has been shutdown. This includes historical retrievals and archive management.

The next section explains simulation in CCTK. The different ways in which to simulate data are presented. The CCTK Tcl simulator is introduced and several simple examples are explained.

The last two sections are reference sections. Manual pages, file specifications, and language definitions are all present in this section.

A glossary of commonly used terms, phrases, and acronyms is included at the end of the manual.

1.2 CCTK Documentation Guide

This manual is one of a set of five manuals included in the CCTK product suite, illustrated in Figure 1-1 below. The manuals are designed to address the needs of particular classes of CCTK users such as installers, administrators, and operators.

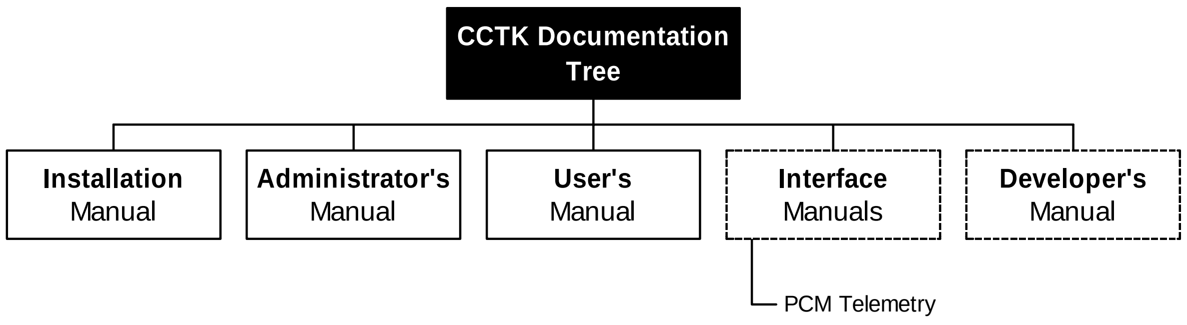


Figure 1-1. CCTK Documentation Tree. Dashed boxes indicate manuals for optional CCTK software products.

Release notes provide supplemental information to this manual. Please refer to the RELEASE_NOTES text file in the root directory of the CCTK CD.

- ***CCTK Installation Manual*** – Describes the steps needed to install CCTK client and CCTK server software from the CCTK CD. Installation on Microsoft Windows® and UNIX® systems is described.
- ***CCTK Administrator's Manual*** – Describes setup, configuration, and administration of CCTK. The material addresses software setup, user administration, project configuration, hardware configuration, and other topics related to administration of a CCTK-based system.
- ***CCTK User's Manual*** – Includes an overview of the CCTK architecture and a complete description of how to use CCTK for operations.
- ***CCTK PCM Telemetry Interface Reference Manual*** – Describes setup, configuration, and use of the optional PCM Telemetry Interface software module.
- ***CCTK Developer's Manual*** – Describes the optional CCTK application development environment and how to use the CCTK Developer's kit to extend and customize CCTK to a particular application or operation with your own software.

1.3 Reference Documentation

The following documents complement the material presented in this manual:

- UNIX System Administration Reference
- C Shell Reference
- Bourne Shell Reference
- *XML Unleashed – From Knowledge to Mastery*, Sams Publishing, Indianapolis, IN, 2000.
- Tcl/Tk is a freely available, platform independent, language environment developed by John Ousterhout. Additional information is available at http://www.neosoft.com/tcl.

1.4 CCT Help Desk

The CCT Help Desk Phone Number is (321) 264-1193 (available 8:00 am to 5:00 pm, U.S. Eastern Time, Monday through Friday, excluding holidays).

CCT provides free unlimited telephone and e-mail support for 30 days after purchase as part of our standard warranty. Customers who purchase the technical support upgrade option have unlimited telephone or email support for one year after purchase or renewal. For questions or comments about CCTK or other CCT products or services please call (321) 264-1193 and ask to speak to a customer representative. For e-mail information, contact info@cctcorp.com or visit our website at http://www.cctcorp.com.

It is important when calling for help with a problem to preserve the project environment as much as possible to allow CCT engineers to efficiently diagnose the problem. If at all possible, leave the project and operation in its current state at the time of the problem and call the Help Desk. If it is not possible to stop and troubleshoot during an operation, please perform the following steps:

1. Completely document all of the symptoms of the problem and as much of the state and configuration of the project as possible.
2. Copy the KPATH directory to a location outside of the project environment (e.g., /tmp/*projectname/date*/).
3. Note the time and date of the problem.

Additional help may be found at the CCT Technical Support web site http://www.cctcorp.com/support. Also, please review the RELEASE_NOTES file in the root directory of the CCTK CD for additional information and instructions.

2 USER ENVIRONMENT

This section discusses the configuration of the user environment on the CCTK server. Many of the graphical user interfaces used by CCTK users execute on the CCTK server which is a UNIX system and are displayed on the user’s client system using features of the X Window System. Therefore, most users of CCTK must have a correctly configured user account on the CCTK server. If a user only wishes to use the remote user applications (such as *StripChart* or *T-Zero*™) then no account is needed on the server.

This section does not discuss user administration for UNIX systems, please reference the appropriate operating system documentation for information on creating and administrating users on the UNIX system. It also does not discuss configuration of the user environment on the CCTK display client. CCTK requires no special configuration of the user environment on the display client. It is up to the administrator of the display clients to create a setup applicable to the local user’s needs.

Configuration of the user’s environment is accomplished through a series of environment variables. The environment variables must be set correctly or CCTK will not operate properly. CCTK requires additions to most of the standard UNIX environment variables (i.e. *PATH*, *LD_LIBRARY_PATH*, etc.). The user’s shell is typically responsible for setting environment variables on a UNIX system. For instructions on setting environment variables, see the documentation associated with the user’s shell.

2.1 Automated Configuration

During installation, the install script creates two shell scripts that contain all of the necessary environment variables to run CCTK: cct_env.sh is for users who prefer to use a Bourne-type shell, and cct_env.csh is for users who prefer to use a C type shell. Both of these scripts are located in CCTK installation directory (“CCT_HOME”) under etc. CCT recommends placing the environment variable configuration within the startup scripts associated with the user’s shell. If this is done, the CCTK environment will be established when a user logs into the CCTK server. Table 2-1 summarizes the shell type, script location, and source command reference for the user configuration scripts.

Table 2-1: User environment configuration scripts

Shell Type	Script Location	Command Reference
Bourne	*${CCT_HOME}/etc/cct_env.sh*	. *${CCT_HOME}/etc/cct_env.sh*
C	*${CCT_HOME}/etc/cct_env.csh*	source *${CCT_HOME}/etc/cct_env.csh*

2.2 Manual Configuration

The shell scripts described in the previous section should correctly configure the user environment a majority of the time. Under certain custom configurations, it may be necessary to manually set all of the environment variables needed by CCTK. When the user is extending CCTK using the development option, there is a good chance that the standard configuration scripts will be inadequate. The *CCTK Developer's Manual* provides additional information on configuration of the user environment to support development. Table 2-2 lists the environment variables associated with CCTK.

Table 2-2: Required user environment variable listing

Environment Variable Name	Value
CCT_HOME	Set to CCT installation directory
GLG_HOME[1]	Set to GLG installation directory
GLG_PALETTES_LOCATION[1]	Set to *${GLG_HOME}/widgets*
PATH[2]	Append *${CCT_HOME}/bin* Append *${GLG_HOME}/bin*
LD_LIBRARY_PATH[3,4]	Append *${CCT_HOME}/lib*
TCLLIBPATH	Append *${CCT_HOME}/lib*
DTDPATH	Set to *${CCT_HOME}/include/dtd*
CCTKPROJECTLABEL	(undefined by default)
KPATH	(undefined by default)

Most of the environment variables listed above are "standard" UNIX environment variables and are explained in the appropriate UNIX documentation. The several that are specific to CCTK are discussed in the following paragraphs.

CCT_HOME and *GLG_HOME* are simply used to locate the installed components in the directory tree. *GLG_PALETTES_LOCATION* is a special GLG environment variable. Please reference the GLG documentation for more information on it. *DTDPATH* is a special CCTK environment variable that allows CCTK to locate the DTD's used by the system.

CCTKPROJECTLABEL is an environment variable. It is not set by default. When set, *CctkClient* will use the text of this variable as a replacement for the term "Project label" in its display. *CCTKPROJECTLABEL* allows the project administrator to customize the wording of the display for a particular domain. For example, "Mission" may be appropriate in the launch processing domain, while "Experiment" may be more appropriate when conducting basic life science research.

KPATH is a special environment variable that points to the project directory of the project the user would like to use. *KPATH* is used by every CCTK process to locate the project directory

[1] Only required when using the GLG Toolkit.

[2] When using IRIX, it is important that /usr/freeware/bin is added to the path.

[3] Under IRIX, LD_LIBRARY_PATH should be changed to LD_LIBRARYN32_PATH.

[4] When using IRIX, it is important that /usr/freeware/lib is added to the library path.

and thus, resources associated with a particular project. *KPATH* typically needs to be set if the user is going to be performing operations from the UNIX command line. Command line operations are typically required during project administration and application development. If a user uses the *CctkClient* application, *KPATH* is set automatically when the user selects “Open Project.” It is not necessary to set *KPATH* for users who use *CctkClient*. However, if *KPATH* is set when *CctkClient* initializes, it will open that project by default.

The following listing shows an example environment configured for an IRIX system with CCTK installed in the /opt/cct directory and GLG installed in the /opt/glg directory. This listing is based upon the environment variables setup by the automatic configuration scripts found in the *${CCT_HOME}/etc* directory.

Listing 2-1: Sample user environment using automatic configuration scripts (IRIX)

```
CCT_HOME=/opt/cct
GLG_HOME=/opt/glg
GLG_PALETTES_LOCATION=/opt/glg/widgets
PATH=/usr/freeware/bin:/usr/local/bin:/opt/cct/bin:/opt/glg/bin:/usr/sbin:/usr/bsd:/sbin
:/usr/bin:/usr/bin/X11:/usr/etc
LD_LIBRARYN32_PATH=/usr/freeware/lib:/usr/local/lib:/opt/cct/lib
TCLLIBPATH=/usr/local/lib /opt/cct/lib /usr/freeware/lib
DTDPATH=/opt/cct/include/dtd
```

3 PROJECT CONFIGURATION

The project administrator can customize CCTK to meet the needs of a particular application. This customization takes place in a series of configuration files located in the project directory. Most of these configuration files are XML. A basic understanding of XML is helpful in comprehending the CCTK project configuration files. It is possible (and likely) for multiple projects to exist on a CCTK server. It is even possible for a single CCTK server to execute multiple projects concurrently.

A consistent naming convention is used for the elements and attributes in the CCTK XML configuration files. The name of an XML element always begins with a capital letter and has each letter of each individual word capitalized. For example, the following items are all elements in the CCTK configuration files.

- *ProjectConfiguration*
- *NoticeDescriptor*
- *AnalogMeasurement*

The name of an XML attribute always begins with a lower case letter and has each letter of each individual word capitalized after the first. For example, the following items are all attributes in the CCTK configuration files.

- *waitFor*
- *minimumSize*
- *minimumSids*

3.1 Project Directory

All information associated with a CCTK project is stored in a single directory. This project directory, known as *KPATH*, contains both project configuration files and files dynamically generated during project execution. Configuration files are explained in this chapter. Files generated during project execution are explained in Section 4. The project directory should reside on a local file system rather than a network file system. Archive bandwidth limitations and file-locking problems may exist if a remote file system is used.

For proper operation of CCTK applications, it is advisable to set the environment variable *KPATH* to point to the project directory containing the project that is being used. The environment variable *KPATH* directs CCTK processes to the correct directory for obtaining important project resources. If *KPATH* is not set, then a command line argument can be used to pass the project directory location to CCTK processes. *KPATH* is automatically set by *CctkClient* whenever a user performs an “Open Project” operation. Therefore, it is not necessary to set *KPATH* for users interacting with *CctkClient*.

It is also important to correctly set the permissions on the project directory. If a user of the system has read permissions to the project directory then they will be able to attach to the project, view data, and issue commands. If they have write permissions to the project directory, they will be able to change the project configuration. If you do not wish for all users of the system to have access to the project, then be sure to restrict access to the directory using the standard UNIX permission utilities. Additional UNIX permissions can be set on the individual files within the project configuration directory to further restrict operations. These permissions will be covered in subsequent sections.

The following configuration files reside in the project directory:

- Project Configuration File
- Database Configuration File(s)
- Archive Configuration File

The following sections further explain these configuration files.

3.2 Project Configuration File

CCTK configuration is defined in the project configuration file. The project configuration file allows the user to customize the following aspects of CCTK :

- Project Name and Description
- Startup Configuration
- *CctkClient* Configuration

Like many of the CCTK configuration files, the project configuration file is based upon XML. The DTD describing the project configuration file can be found in *${CCT_HOME}/include/dtd/project_config.dtd.*

It is important for the project administrator to ensure that the permissions on the project configuration file are set correctly. Table 3-1 describes how each of the primary UNIX permission bits is used in relation to the project configuration file.

Table 3-1: Project configuration file permissions matrix

Permission Bit	Purpose
read (r)	If a user has read permissions on the project configuration file, they may attach to the project. When a user attaches to a project, they can view data within the project and execute commands associated with the project.
write (w)	If a user has write permissions on the project configuration file, they may change the project configuration file.
execute (x)	If a user has execute permissions on the project configuration file, they may start/stop the project.

3.2.1 Basic Structure

Listing 3-1 shows the basic structure of the project configuration file.

Listing 3-1: Basic XML structure of project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
	<Description>A description of the overall project.</Description>
	<StartUp defaultMode="Default Mode">
		[… Elements Removed …]
	</StartUp>
	<CctkClient>
		[… Elements Removed …]
	</CctkClient>
</ProjectConfiguration>
```

As with all XML files, the version is contained on the first line. Immediately following the version is the document type. Project name, which appears on the *CctkClient* user interface is defined via the attribute *project_name* of the document element. An optional project description, which also appears within *CctkClient*, is defined via the *Description* element. Next, each project configuration file must have a *StartUp* element defining the resources needed to startup the project. Finally, an optional *CctkClient* element defines the configuration of the *CctkClient* user interface.

3.2.2 Specifying Startup Resources

The StartUp element is used to define the resources associated with the CCTK project. These resources include:

- Modes
- Status Table
- Configuration Database Reference
- Channels
- Processes

Each of these resources is further discussed in the following sections.

3.2.2.1 Defining Modes

Each CCTK project consists of one or more modes configured by the project administrator. The *Mode* element is used to define these modes. Modes are used to configure the required resources during project execution. The user is able to select a mode when starting a project. Each mode is given a unique name (via the *name* attribute) and contains a set of resources to create when a project is started (via the elements). Each mode may also inherit another mode (via the *inherits* attribute). When a mode inherits another mode, the project is started with all of the resources associated with both modes. Listing 3-2 shows a project configuration file with three modes configured.

Listing 3-2: Defining modes within the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
	<Description>A description of the overall project.</Description>
	<StartUp defaultMode="Mode C">
		<Mode name=”Mode A”>
			[… Elements Removed …]
		</Mode>
		<Mode name=”Mode B” inherits=”Mode A”>
			[… Elements Removed …]
		</Mode>
		<Mode name=”Mode C” inherits=”Mode B”>
			[… Elements Removed …]
		</Mode>
	</StartUp>
	[… Element Removed …]
</ProjectConfiguration>
```

In the above example, if the user starts the project in mode A, then only the resources associated with mode A will be created. If the user starts the project in mode C, then the resources associated with mode C, mode B, and mode A will be created. There is no limit to the number of modes associated with each project.

3.2.2.2 Creating the Status Table

Each active CCTK process has an entry in the CCTK status table. The status table is created at startup and preallocates a maximum number of processes that may be associated with a single project. The element *StatusTable* with the attribute *entries* is used to specify the number of entries to associate with the status table. It is important to allocate enough status table entries for all of the processes that may be active at any one time during project execution. The following formula provides a basic algorithm for estimating the number of status table entries needed:

$$entries = (simultaneous\ clients + CCTK\ system\ processes) * 1.1$$

The *entries* attribute is used to specify the number of status table entries associated with this mode as shown in Listing 3-3.

Listing 3-3: Defining the status table within the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
```

```
       <Description>A description of the overall project.</Description>
       <StartUp defaultMode="Mode C">
              <Mode name=”Mode A”>
                     <StatusTable entries=”100” />
                     [… Elements Removed …]
              </Mode>
       </StartUp>
       [… Element Removed …]
</ProjectConfiguration>
```

The *StatusTable* element may only be specified once for each mode. The total status entries allocated for a project is the sum of the entries for all modes.

3.2.2.3 Configuration Database References

Each project mode can reference one or more configuration database files. Each configuration database file is parsed in turn to create the configuration database for the project. The *ConfigDb* element and the *file* attribute are used to specify configuration database files. Details on defining the contents of the configuration database files are discussed in Section 3.4. If the path to the file is not absolute, then they are referenced from the project directory.

Listing 3-4: Defining configuration database file within the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
       <Description>A description of the overall project.</Description>
       <StartUp defaultMode="Mode C">
              <Mode name=”Mode A”>
                     <ConfigDb file=”project_db_1.xml” />
                     <ConfigDb file=”project_db_2.xml” />
                     [… Elements Removed …]
              </Mode>
       </StartUp>
       [… Element Removed …]
</ProjectConfiguration>
```

3.2.2.4 Creating Channels

CCTK processes use channels as an inter-process communications mechanism within CCTK. Each CCTK project must define a set of standard channels. In addition, the user may optionally define additional channels for passing data around the system. Additional channels are typically used to pass data between interface processes and core CCTK processes. The attributes used to define a channel are shown in Table 3-2.

Table 3-2: *Channel* element attributes

Attribute	Description
name	Name of the channel. This is typically an input argument to CCTK processes.
size	Size of the channel in bytes. This value should be at least 5000 bytes. It should be increased for channels that carry a lot of traffic. The higher value allows for a larger buffer between processes and helps eliminate context switching between applications, but increases latency.
count	*count* should only be used when specifying the special channel "NEXT_AVAIL_CHAN". CCTK processes can request an unnamed channel via the channel subsystem. When an unnamed channel is requested by a process, one of the channels from the "NEXT_AVAIL_CHAN" list is selected. This count indicates how many channels should be created for this dynamic pool.

The *Channel* element is used to define channels. The *Channel* element must appear within a *Mode* element. Listing 3-5 shows the definition of the standard CCTK channels and ten NEXT_AVAIL_CHAN.

Listing 3-5: Defining channels within the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name="Project Name">
        <Description>A description of the overall project.</Description>
        <StartUp defaultMode="Mode C">
                <Mode name="Mode A">
                        <Channel name="STDMSG" size="40000" />
                        <Channel name="STDRESP" size="40000" />
                        <Channel name="STDDIST" size="40000" />
                        <Channel name="STDSTATUS" size="40000" />
                        <Channel name="STDARCH" size="40000" />
                        <Channel name="NEXT_AVAIL_CHAN" size="5000" count="10" />
                        [… Elements Removed …]
                </Mode>
        </StartUp>
        [… Element Removed …]
</ProjectConfiguration>
```

It is important to note that each CCTK project must define the standard channels listed above (STDMSG, STDREP, STDDIST, STDSTATUS, and STDARCH). Another requirement is to define a minimum number of NEXT_AVAIL_CHAN.

3.2.2.5 Executing Processes

The final item specified within the *Mode* element of the project configuration file is the *Execute* element. The *Execute* element specifies the commands to execute to complete the project startup. Typically, each CCTK project will execute at least 3-5 and possibly as many as 30-50 different processes that will perform the actual work associated with the project. Each process executed has a different job (such as message passing, data processing, user interface, etc.) Each mode has a list of processes to execute. When the project is started, *Execute* elements are executed in the order they appear in the project configuration file, with the parent modes *Execute* elements being executed first.

Each *Execute* element has three possible attributes as described in Table 3-3.

Table 3-3: *Execute* element attributes

Attribute	Description
waitFor	waitFor can take on the value of “running”, “exit”, or “period”. If *waitFor* is “running”, then the next execute statement will not be executed until the status of the current statement is set to “running”. If *waitFor* is “exit”, then the next execute statement will not be executed until the current statement exits. If *waitFor* is “period”, then the next execute statement will execute after a certain period of time passes. Use *waitFor* “period” with a period of “0” to not wait at all. Please see the text at the end of this section for important information on the *waitFor* “running” option.
period	This option has a different meaning depending on the value of *waitFor*. If *waitFor* is “running” or “exit”, then *period* represents a timeout. This is the longest time that will elapse until a failure condition is assumed. If *waitFor* is “period”, then period is the amount of time to wait. Period is given in seconds and defaults to 20.
critical	Valid values are “YES” or “NO”. If an “execute” statement is marked as critical, project startup will fail if the execute statement does not exit with a success value. In addition, the process will be monitored and a system message will be generated if the process executed stops updating its health count.

Listing 3-6 shows examples of several different execute elements.

Listing 3-6: Defining *Execute* elements within the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
      <Description>A description of the overall project.</Description>
      <StartUp defaultMode="Mode C">
            <Mode name=”Mode A”>
                  <Execute waitFor=”running” critical=”YES”>
                        DataProc –t mdt -i STDRESP</Execute>
                  <Execute waitFor=”running” period=”10” critical=”YES”>
                        Message</Execute>
                  <Execute waitFor=”period” period=”5” critical=”NO”>
                        UserCommand1</Execute>
                  <Execute waitFor=”exit” period=”30” critical=”NO”>
                        UserCommand2</Execute>
            [… Elements Removed …]
            </Mode>
      </StartUp>
      [… Element Removed …]
</ProjectConfiguration>
```

For more information on the available CCTK processes, their options, and instructions on using them, reference Section 3.5. There are several processes that are required for CCTK operation. Section 3.5 also covers those processes that are required versus those that are optional.

The use of the waitFor “running” option requires that the application be designed to use the CCTK process state reporting mechanisms. CCTK provides a set of API calls that allow a process to report its current state to the system. The waitFor “running” option uses these API calls to determine when a process transitions to the “running” state and thus, when it is safe to continue with execution of the next statement. It is important that waitFor “running” is only used with properly designed CCTK processes. All of the core CCTK processes are designed to support the waitFor “running” option. User developed applications may support the waitFor “running” option if they were designed with support in mind. All UNIX applications and third party applications will not support the waitFor “running” option and therefore must use one of the other options. If, when using the waitFor “running” option, the project initialization always fails on the same processes with a timeout, try changing it to a different option.

3.2.3 Specifying CctkClient Resources

The project configuration file allows the project administrator to exercise global control over the contents of the display tree list.

The basic structure of the project configuration file *CctkClient* element is shown in Listing 3-7.

Listing 3-7: Basic XML structure of *CctkClient* configuration in project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
       <StartUp>
              [… Elements Removed …]
       </StartUp>
       <CctkClient>
              <DisplayTree>
                     [… display tree elements go here …]
              </DisplayTree>
       </CctkClient>
</ProjectConfiguration>
```

Table 3-4 lists the valid display tree elements and provides a brief description of each.

Table 3-4: Valid display tree elements

Type	Element Name	Description
Interface	*DisplayTreeInterfaces*	Shows the list of active interfaces associated with this project. The list of interfaces is dynamically generated based upon the database configuration for the project. The system must be “UP” before the list is generated.

Type	Element Name	Description
Execute	*DisplayTreeExec*	Allows the execution of an arbitrary command. This is primarily used to support applications external to *CctkClient*. Examples include *MonCon*, *MeasMon*, and other graphical display items. This does NOT include display applications executed on the display client such as *T-Zero* and *StripChart*.
State History	*DisplayTreeStateHistory*	Shows the state history associated with this project.
List	*DisplayTreeList*	A list of other display tree items, including other lists. Lists allow the project administrator to group items within the display tree.

Each of the above is represented by an element in the display tree. Usage of these elements is described in the following sections.

All display tree elements share a common set of sub-elements and attributes. Table 3-5 lists these common attribute and elements.

Table 3-5: Common display tree attributes and elements

Name	Type	Description
name	attribute	*name* is the name of the item. The name will be displayed in the *CctkClient* display and will be seen by the user.
ToolTip	element	The contents of the *ToolTip* element will be displayed when the user requests the tooltip for an item in the display tree. A tooltip is displayed when the user leaves his/her mouse over a display tree item for more than a few seconds. The tooltip element typically contains a brief, single line description of the item.
WhatsThis	element	The contents of the *WhatsThis* element will be displayed when the user requests the "What's This" help for an item in the display tree. This occurs when the user selects the "Help→What's This" menu item and clicks on an item in the display tree. The *WhatsThis* element typically contains a longer, multi-line description of the item.

3.2.3.1 Creating an Interface List in the Display Tree

Listing 3-8 shows an example of adding an interface list to the *CctkClient* display tree. Note that the *ToolTip* and *WhatsThis* elements are shown in this example.

Listing 3-8: Defining a *DisplayTreeInterfaces* element in the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
	<StartUp>
		[… Elements Removed …]
	</StartUp>
	<CctkClient>
		<DisplayTree>
			<DisplayTreeInterfaces name=”Interfaces”>
				<ToolTip>Valid interfaces for the current project and
mode.</ToolTip>
				<WhatsThis>This displays a dynamically generated list of
```

```
interfaces that are valid for the current project and mode. This list can change during
project execution as projects are started and stopped.</WhatsThis>
                    </DisplayTreeInterfaces>
                    [… other display tree elements …]
             </DisplayTree>
      </CctkClient>
</ProjectConfiguration>
```

Whenever an interface is developed within CCTK, one of the components of interface development is a control widget. The control widget displays important health and status about the interface as well as provides control for issuing commands to the interface. When a project starts, an internal mapping is created between the valid interfaces for the current project and mode and their respective control widgets. The “interface list” display tree element instructs the display tree to search this internal mapping and place all valid control widgets into the display tree.

The interface list is dynamically generated when a project executes. Therefore, the list will remain empty until a project is executed for the first time. Once a project initializes, the internal database will be searched and any valid port with a control widget specified will be placed into the interfaces display tree list automatically. When a project is stopped, this list is not cleared. If a different project or mode within the same project is started which has different interfaces defined, those will also be added to the interfaces display tree list. Only those interfaces associated with the current project will be active however. This allows the user to switch between two projects and maintain the interface displays. To clear the display tree list, simply select Close from the *CctkClient* menu.

3.2.3.2 Launching External Applications from the Display Tree

Listing 3-9 shows an example of adding an external application to the *CctkClient* display tree. Note that the *ToolTip* and *WhatsThis* elements are also shown.

Listing 3-9: Defining a *DisplayTreeExec* element in the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
      <StartUp>
             [… Elements Removed …]
      </StartUp>
      <CctkClient>
             <DisplayTree>
                    <DisplayTreeExec name=”Weather” exec=”MonCon weather.glg”>
                           <ToolTip>Launch the Weather Display</ToolTip>
                           <WhatsThis>This will item will launch the Weather Display using
the CCTK MonCon application.</WhatsThis>
                    </DisplayTreeExec>
                    [… other display tree elements …]
             </DisplayTree>
```

```
       </CctkClient>
</ProjectConfiguration>
```

The *DisplayTreeExec* element includes a required *exec* attribute. The *exec* attribute specifies the command to run. The command is passed to Bourne shell for execution. Therefore, any valid Bourne shell syntax is valid within the *exec* attribute.

3.2.3.3 Adding State History to the Display Tree

Listing 3-10 shows an example of adding a state history view to the *CctkClient* display tree. Note that the *ToolTip* and *WhatsThis* elements are also shown.

Listing 3-10: Defining a *DisplayTreeStateHistory* element in the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
       <StartUp>
              [… Elements Removed …]
       </StartUp>
       <CctkClient>
              <DisplayTree>
                     <DisplayTreeStateHistory name=”State History”>
                            <ToolTip>Show the state history widget.</ToolTip>
                            <WhatsThis>When selected, this list item will display the state
history widget. The state history widget shows the state transitions associated with the
current CCTK project.</WhatsThis>
                     </DisplayTreeStateHistory>
                     [… other display tree elements …]
              </DisplayTree>
       </CctkClient>
</ProjectConfiguration>
```

Only one *DisplayTreeStateHistory* element should be added to the display tree list. For more information on the purpose of the state history view and how to use it, please reference the *CCTK User’s Manual*.

3.2.3.4 Creating Lists in the Display Tree

Since the display tree is a “tree” type structure, it is possible to add a new “branch” to the display tree that is a container for other display tree elements. Each “branch” is simply called a new display tree list. Listing 3-11 shows an example of adding a list to the *CctkClient* display tree. In this example, a display tree list is used to group together a set of electrical displays.

Listing 3-11: Defining a *DisplayTreeList* element in the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
      <StartUp>
            [… Elements Removed …]
      </StartUp>
      <CctkClient>
            <DisplayTree>
                  <DisplayTreeList name=”Electrical Displays”>
                        <DisplayTreeExec name=”Electrical Display 1”
                                    exec=”MonCon elec_display_1.glg”/>
                        <DisplayTreeExec name=”Electrical Display 2”
                                    exec=”MonCon elec_display_2.glg”/>
                  </DisplayTreeList>
                  [… other display tree elements …]
            </DisplayTree>
      </CctkClient>
</ProjectConfiguration>
```

Lists can be inserted into other lists creating a hierarchy type structure. Listing 3-12 shows an example of a nested list. This example is an extension of the above where there are multiple electrical subsystems and each subsystem has its own nested list.

Listing 3-12: Defining a nested *DisplayTreeList* element in the project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
<ProjectConfiguration project_name=”Project Name”>
      <StartUp>
            [… Elements Removed …]
      </StartUp>
      <CctkClient>
            <DisplayTree>
                  <DisplayTreeList name=”Electrical Displays”>
                        <DisplayTreeList name=”Stage 1 Electrical Subsystem”>
                              <DisplayTreeExec name=”Electrical Display 1”
                                          exec=”MonCon stage_1/elec_display_1.glg”/>
                              <DisplayTreeExec name=”Electrical Display 2”
                                          exec=”MonCon stage_1/elec_display_2.glg”/>
                              [ … other display tree elements …]
                        </DisplayTreeList>
                        <DisplayTreeList name=”Stage 2 Electrical Subsystem”>
                              <DisplayTreeExec name=”Electrical Display 1”
                                          exec=”MonCon stage_2/elec_display_1.glg”/>
                              <DisplayTreeExec name=”Electrical Display 2”
                                          exec=”MonCon stage_2/elec_display_2.glg”/>
                              [ … other display tree elements …]
                          </DisplayTreeList>
                          [… other display tree elements …]
                      </DisplayTreeList>
                      [… other display tree elements …]
              </DisplayTree>
        </CctkClient>
</ProjectConfiguration>
```

3.2.4 Standard Project Configuration File Template

A sample project configuration file can be found at:

${CCT_HOME}/examples/template_project_config.pcml

This sample can be used as a template when creating new project configurations. Some key points to note regarding this sample file:

- Two modes are defined; “Core CCTK” and “Custom CCTK”.
- “Core CCTK” contains the minimum resources necessary to run CCTK.
- “Custom CCTK” contains comments on how to extend the configuration to perform additional tasks.
- A simple display tree is defined. It contains the most common elements as well as some comments for adding additional elements.

Additional examples of project configuration files can be found in the *${CCT_HOME}/projects* directory. Within this directory are several example projects, all of which have a project configuration file.

3.3 Archive Configuration File

The archive configuration file is used to override the default configuration of the CCTK archive subsystem. The archive configuration file, typically located in the project directory, is a simple ASCII text file. It contains a list of configuration parameters and the appropriate value for each parameter. There is one entry per line. Each entry is formatted as follows:

<parameter_name> <parameter_value>

The file can be edited to tune execution of the archive subsystem. Not all parameters need to be specified in the archive configuration file. In the absence of a configuration file or any one parameter, the default value for that parameter is used. In many cases, the larger the value for a parameter, the more memory the archive subsystem will consume.

The archive configuration file must be passed on the command line to the *TamArs* process (one of the primary processes of the archive subsystem). Although it is typically called ars_config.d, the user can choose any name for the file. The *TamArs* process reads the archive subsystem configuration file. If a configuration file name is specified on the *TamArs* command line, it is read in during *TamArs* initialization. A configuration file must be specified on the command line to override the default values.

Table 3-6 presents the valid parameters for the archive configuration file along with a description, the possible range of values, and the default value.

Table 3-6: Archive configuration parameters

Parameters	Description	Range	Default Value
max_SIDs	Max data ids (SIDs) to be archived. This should be the sum of the measurements, commands, and notices within the system that will be archived. Reducing the size of this value allows the archive subsystem to use less memory. Increasing the size of this value causes the archive subsystem to use more memory.	1000..10000	5000
tam_size	Size (bytes) of the temp archive (disk). If the temporary archive exceeds this size, the archiving will stop.	500KB..8GB	1GB
data_compress	Compression of archived data. Indicates if the internal data compression algorithm is off or on. This significantly reduces disk space at the cost of minimal processing.	ON | OFF	ON
fd_refresh	Refresh cyclic interval (minutes). At this indicated interval, the archive subsystem will automatically write the last known value of all measurements not received in the previous interval out to disk.	1..10	10
tam_path	Full path (directory) containing the path where the archive subsystem will store the archive files.	n/a	(project directory)
tam_auto_start	Automatic activation of TAM recording startup. This indicates whether the archive subsystem should be activated on system startup or if it should wait to be activated by a user.	ON | OFF	ON

Listing 3-13 shows a sample archive configuration file.

Listing 3-13: Sample archive configuration file

```
max_SIDs 1000
tam_size 2048000000
tam_path /tmp/TAM
```

The first line sets the maximum number of system identifiers (SIDs) to be archived to 1000. The second line sets the size of the archive files to 2GB. The third line moves the archive files from their default location in the project directory to the tmp directory of the system. Note that the archive size may be limited by the maximum file size of your operating system.

3.4 Configuration Database

The CCTK configuration database stores the table and descriptor configuration for a project. Section 3.2.2.3 discusses how to reference configuration database files from within the *Mode* elements of the project configuration file. Upon project startup, *ProjectManager* will parse and load the tables and descriptors defined in all of the configuration database files associated with a particular mode. It is possible, and usually desirable, to break the CCTK configuration database into multiple files to facilitate:

- Organization: Related configuration information can be contained in separate files making for better organization and easier maintenance.
- Selective Loading: With the database broken into multiple files, modes can be configured to selectively load only the tables/descriptors necessary for a particular application.
- Easier Troubleshooting: Multiple files allow errors to be isolated to smaller blocks of data and thus, it is simpler to find problems.

Since the format of the configuration database files is XML, a DTD is used to describe the file syntax. The configuration database DTD file can be found at *${CCT_HOME}/include/dtd/xcdb_config.dtd*. The following sections provide a detailed description of the valid elements and attributes that can be used in defining a CCTK configuration database.

3.4.1 Basic Structure

Listing 3-14 shows the basic structure of the configuration database file.

Listing 3-14: Basic XML structure of configuration database file

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase shlibs=”[… dynamic libraries …]”>
      [… table and table group elements …]
      [… descriptor and descriptor group elements …]
</ConfigurationDatabase>
```

As with all XML files, the version is contained on the first line. Immediately following the version is the document type. *ConfigurationDatabase* is the recommended document type for most CCTK configuration database files. Within the *ConfigurationDatabase* element, the user defines any number of table, table group, descriptor, and descriptor group element types. Section 3.4.11 provides information on custom databases in which the user may extend the database to include additional custom descriptors. In these custom cases, the document type will change.

There are several descriptors within the CCTK configuration database that can reference other descriptors. For example, an analog measurement references both a conversion and an exception. When a reference is necessary, it can be declared one of two ways:

- In-line: In this case, the referenced item is in-lined with the parent item. The in-lined item is fully declared with the XML element of the parent.
- Reference: In this case, only the name of the referenced item is placed within the parent item’s XML element. It is up to the user to declare an item of that type elsewhere within the configuration database.

Examples of in-line and reference declarations are presented in several of the following sections.

3.4.2 Defining Tables

Each CCTK configuration database must define a set of real-time tables. Each real-time table stores descriptors of related information. The real-time tables must be defined in the configuration database before any descriptors that will be placed into them. It is permissible to define a table, then all of the descriptors associated with that table, then define the next table, then all of the descriptors associated with that table. Another technique is to first define all of the tables and then define all of the descriptors.

The *Table* element accepts several attributes, the most common of which is name. The other attributes control storage aspects of the table. Under normal circumstances, CCTK will automatically size the tables and these attributes are not needed.

Table 3-7 lists all of the attributes associated with the *Table* element. The shared column indicates if the element or attribute can be used when defining a *TableGroup*. Groupings are an advanced concept and further defined in Section 3.4.11.

Table 3-7: Table elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
name	attribute	string	no	yes	–
	Name of the table, must be unique within a CCTK project.				
Description	element	string	yes	no	(empty string)
	A description of the table.				
size	attribute	integer	yes	no	(see below)
	If *size* is specified, the table will always be that size. The automated table sizing is not performed and the *minimumSize* and *percentSpareSize* attributes are ignored.				
minimumSize	attribute	integer	yes	no	25,000 (see below)
	The table will never be sized smaller than this value when its size is calculated. If the calculated size is less than *minimumSize*, it will be increased to minimum size.				
percentSpareSize	attribute	real	yes	no	10%
	After calculating the necessary size of the table, the size will be increased by this percentage which will be used as spare space for run-time updates of the table.				
sids	attribute	integer	yes	no	(see below)
	If *sids* is specified, the table will always be allocated to hold this number of SID's. The automated table sizing is not performed and the *minimumSids* and *percentSpareSids* are ignored.				
minimumSids	attribute	integer	yes	no	(see below)
	The table will be sized to hold at least *minimumSids* when the number of SID's is calculated. If the calculated SID's is less than *minimumSids*, it will be increased to hold the value specified.				
percentSpareSids	attribute	real	yes	no	10%
	After calculating the necessary number of SID's for the table, the total will be increased by this percentage which will be used as spares for run-time updates of the table.				

Listing 3-15 provides an example of defining several tables.

Listing 3-15: Configuration database table example

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <Table name=”Example Table 1” />
      <Table name=”Example Table 2” />
      <Table name=”Example Table 3” percentSpareSpace=”20” percentSpareSids=”20”>
            <Description>Description of Table 3</Description>
      </Table>

</ConfigurationDatabase>
```

It is important to note that if an element or attribute is not specified for a table, then the system will use a default value. Most elements/attributes associated with tables (and the other configuration database items) are optional. It is only necessary to use the attribute/element when a value other than the default is desired.

CCTK requires a set of tables be defined with a particular name and in a particular order. Table 3-8 lists the required table names and ordering.

Table 3-8: Required CCTK tables

Table Name	Description
ndt	Notice Descriptor Table: Contains information, including name and attributes, associated with the notices defined for the running CCTK project.
mdt	Measurement Descriptor Table: Contains information, including name and attributes, associated with the measurements, exceptions, and conversions defined for the running CCTK project.
cdt	Command Descriptor Table: Contains information, including name and attributes, associated with the system commands defined for the running CCTK project.
ldt	Link Descriptor Table: Contains information, including name and attributes, associated with external commands and link records that link a measurement with an external interface.
pdt	Port Descriptor Table: Contains information, including name and attributes, associated with a port of an external interface.
bdt	Bus Descriptor Table: Contains information, including name and attributes, associated with a bus of an external interface.

It is possible to define additional tables for a project, but the above tables must be defined first and in the given order. You may wish to define additional tables to hold custom configuration information and/or descriptors related to system customization. Listing 3-16 shows an example definition of the tables listed in Table 3-8.

Listing 3-16: CCTK table definition

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

<Table name="ndt">
      <Description>Notice Descriptor Table.</Description>
```

```
</Table>
<Table name="mdt">
        <Description>Measurement Descriptor Table.</Description>
</Table>
<Table name="cdt">
        <Description>Command Descriptor Table.</Description>
</Table>
<Table name="ldt">
        <Description>Link Descriptor Table.</Description>
</Table>
<Table name="pdt">
        <Description>Port Descriptor Table.</Description>
</Table>
<Table name="bdt">
        <Description>Bus Descriptor Table.</Description>
</Table>

</ConfigurationDatabase>
```

3.4.3 Descriptors

Descriptor is a general term referring to a single, named object within CCTK. Measurements, commands, notices, and other objects are descriptors within CCTK. Most operations within CCTK use a descriptor as the object of the operation. Each descriptor shares several important characteristics:

- Each is contained within a table.
- Each has a unique name within a project.
- Each is given a unique, numbered system identifier (SID).
- Each has a description.

Table 3-9 lists the common elements/attributes applicable to all descriptors.

Table 3-9: Descriptor elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
name	attribute	string	no	yes	–
	Name of the descriptor, must be unique in the CCTK project.				
table	attribute	string	yes	no	(varies by type)
	Table the descriptor will reside in. CCTK will default all descriptors to the correct table. This attribute is not needed in most circumstances. It must correspond to the name of a previously defined table. The default tables for the common CCTK descriptors are listed below in Table 3-10.				
Description	element	string	yes	no	–
	A brief description of the descriptor.				

Table 3-10 lists the types of descriptors that can be defined in CCTK.

Table 3-10: CCTK descriptors

Element Name	Default Table	Section	Description
AnalogMeasurement	mdt	3.4.4.2	Contains the attributes and properties associated with an analog measurement in CCTK.
AnalogException	mdt	3.4.5	Contains the attributes and properties associated with an analog exception in CCTK.
DiscreteMeasurement	mdt	3.4.4.3	Contains the attributes and properties associated with a discrete measurement in CCTK.
DiscreteException	mdt	3.4.5	Contains the attributes and properties associated with a discrete exception in CCTK.
SignedIntMeasurement	mdt	3.4.4.4	Contains the attributes and properties associated with a signed integer measurement in CCTK.
SignedIntException	mdt	3.4.5	Contains the attributes and properties associated with a signed integer exception in CCTK.
UnsignedIntMeasurement	mdt	3.4.4.4	Contains the attributes and properties associated with an unsigned integer measurement in CCTK.
UnsignedIntException	mdt	3.4.5	Contains the attributes and properties associated with an unsigned integer exception in CCTK.
ByteArrayMeasurement	mdt	3.4.4.5	Contains the attributes and properties associated with a byte array measurement in CCTK.
ByteArrayException	mdt	3.4.5	Contains the attributes and properties associated with a byte array exception in CCTK.
StringMeasurement	mdt	3.4.4.6	Contains the attributes and properties associated with a string measurement in CCTK.
StringException	mdt	3.4.5	Contains the attributes and properties associated with a string exception in CCTK.
NoticeDescriptor	ndt	3.4.7	Contains the attributes and properties associated with a notice in CCTK.
PacketDcomDescriptor	ldt	3.4.9	Contains the attributes and properties associated with a decommutation packet in CCTK.
IntegerConversion	mdt	3.4.6.1	Contains the attributes and properties associated with an integer conversion in CCTK.
PolynomialConversion	mdt	3.4.6.2	Contains the attributes and properties associated with a polynomial conversion in CCTK.
PiecewiseLinearConversion	mdt	3.4.6.3	Contains the attributes and properties associated with a piecewise linear conversion in CCTK.

Note that there is no generic “Descriptor” or “Measurement” descriptor element. A generic descriptor (or measurement) is an abstract construct and cannot be specified in the CDB. Examples of defining different descriptor types are shown in the following sections.

3.4.4 Measurements

Measurements are descriptors that store the current value of some item. This item can be a physical item such as a temperature sensor, or a virtual item, such as a state of the processing of some subsystem. There are four primary types of measurements in CCTK:

- Analog: Represents floating point numbers.

- Signed/Unsigned Integer: Represents integer numbers.
- Discrete: Represents items that have two states (such as a switch).
- Byte Array/String: Represents character data as either a list of bytes or a null-terminated string.

Measurements are a type of descriptor so all elements and attributes associated with a descriptor are valid for a measurement. In addition, a measurement has several specific attributes and elements.

Table 3-11 shows the valid elements and attributes for a measurement.

Table 3-11: Measurement elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
rawDataSize	attribute	integer	yes	no	(varies)
	rawDataSize of the measurement rounded to the nearest byte. Default value varies by type. For byte arrays, this is the maximum permissible size of the byte array (i.e. storage space).				
distributeToChannels	attribute	string	yes	no	STDARCH
	Indicates the channels to distribute this measurement. Typically this will always be STDARCH. If it is empty, then no distribution will take place and the measurement will not be archived.				
ProcessingReference	element	n/a	yes	no	(none)
	Defines a reference to a processing module for this measurement. This is exclusive with any in-line processing reference.				
<any processing element>	element	n/a	yes	no	(none)
	Defines a processing module to associate with this measurement in-line. This is exclusive with ProcessingReference.				
StaleProcessing	element	n/a	yes	no	–
	Stale processing configuration for a measurement.				

Processing reference allows a processing module to be associated with this measurement. Processing modules are used to perform conversions between different data types. See Section 3.4.6 for details on specifying processing chains.

Stale processing is a type of measurement processing that applies to all measurements. Stale processing verifies that a measurement is changing as samples are received. If stale processing is enabled, each time a measurement is received, it will be compared to the previous value. If the value has not changed, the stale count will increment. If the stale count exceeds a user defined value, then an exception will be generated and the measurement will be declared stale. Table 3-12 lists the attributes associated with the *StaleProcessing* element.

Table 3-12: *StaleProcessing* elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
active	attribute	true or false	yes	no	true
	Indicates if stale processing is active for a particular measurement.				
count	attribute	integer	yes	yes	(none)
	The number of identical samples to receive before marking a measurement as stale.				

3.4.4.1 Common Measurement Properties

There are several elements that are common to certain types of measurements, but do not apply globally to all measurements. Since these elements do not apply to all measurements, they cannot appear as shared elements within a measurement. However, since they apply to multiple measurement types, it is useful to look at them prior to examining the individual measurement types. The common measurement elements include:

- Significant change
- Maximum change
- Range

The *SignificantChange* element controls the significant change processing associated with either an analog or integer (signed and unsigned) measurement. When significant change is enabled, a measurement will not be processed unless it has changed by a user-defined amount. By enabling significant change, the total number of processed measurements decreases and the processor load can be reduced. Table 3-13 shows the valid attributes for the *SignificantChange* element.

Table 3-13: *SignificantChange* elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
active	attribute	true or false	yes	no	true
	Indicates if significant change processing is active for a particular measurement.				
value	attribute	integer / real	yes	yes	(none)
	The amount a value must change by before it is considered significant and thus processed through the system. Value is an integer if describing an integer measurement; real if describing an analog measurement.				

The *MaximumChange* element controls the maximum change processing associated with either an analog or integer (signed and unsigned) measurement. The maximum change processing algorithm will generate an exception for a measurement when the value of the measurement changes by more than the indicated amount. Some measurements should not change quickly, this generic processing algorithm can be used to detect unwanted changes. Table 3-14 shows the valid attributes for the *MaximumChange* element.

Table 3-14: *MaximumChange* elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
active	attribute	true or false	yes	no	true
	Indicates if maximum change processing is active for a particular measurement.				
value	attribute	integer / real	yes	yes	(none)
	The amount of change the measurement must exceed to force a maximum change exception to be generated. Value is an integer if describing an integer measurement; real if describing an analog measurement.				

The *Range* element controls the generic range checking associated with either an analog or integer (signed and unsigned) measurement. The range processing algorithm will generate an exception for a measurement when the value of the measurement is outside the given range. This range checking is considered the maximum permissible range of the measurement based upon the hardware. CCT recommends setting this range to the precision of the hardware. For example, if the system is processing a 12-bit value, then this range would be set appropriately to cover the entire 12-bit range. In this way, if this exception is generated, it usually indicates a serious problem with the configuration of the system. For user defined ranges based upon certain states of the system, CCT recommends using exceptions (see Section 3.4.5). Table 3-15 shows the valid attributes for the Range element.

Table 3-15: Range elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
active	attribute	true or false	yes	no	true
	Indicates if range processing is active for a particular measurement.				
lower	attribute	integer / real	yes	yes	(none)
	The lower bound for the range-checking algorithm. Value is an integer if describing an integer measurement; real if describing an analog measurement.				
upper	attribute	integer / real	yes	yes	(none)
	The upper bound for the range-checking algorithm. Value is an integer if describing an integer measurement; real if describing an analog measurement.				

3.4.4.2 Defining Analog Measurements

Analog measurements are a type of measurement used to store a floating point value. Analog measurements have the native type of double, which, on most systems, is a 64 bit IEEE floating point value. Conversion routines are used to convert the raw value received from the external interface to the appropriate native type. The *AnalogMeasurement* element is used to declare an analog measurement in the CCTK configuration database. Since analogs are a type of measurement and thus a type of descriptor, all of the elements/attributes associated with measurements and descriptors apply to analogs.

The elements and attributes unique to an analog measurement are listed in Table 3-16

Table 3-16: Analog measurement elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
ExceptionReference	element	n/a	yes	no	(none)
	Defines a reference to an exception for a measurement. This is exclusive with AnalogException.				
AnalogException	element	n/a	yes	no	(none)
	Defines an analog exception to associate with this measurement in-line. This is exclusive with ExceptionReference.				
Units	element	string	yes	no	(none)
	The units associated with this measurement.				
SignificantChange	element	n/a	yes	no	(inactive)
	Defines the significant change properties associated with this measurement.				

Name	XML Type	Data Type	Shared	Required	Default Value
MaximumChange	element	n/a	yes	no	(inactive)
	Defines the maximum change properties associated with this measurement.				
Range	element	n/a	yes	no	(inactive)
	Defines the range properties associated with this measurement.				
InitialValue	element	real	yes	no	0.0
	Defines the initial value associated with this measurement.				

Listing 3-17 provides an example of defining an analog measurement.

Listing 3-17: Configuration database example, defining analog measurements

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <AnalogMeasurement name=”Sample Analog 1” />

      <AnalogMeasurement name="Sample Analog 2" active="true" rawDataSize="8">
            <Description>Description for Sample Analog 2</Description>
            <PolynomialConversion name="Polynomial" table="mdt">
                  <PolynomialCoefficient coefficient="0" value="5" />
                  <PolynomialCoefficient coefficient="1" value="10" />
            </PolynomialConversion>
            <StaleProcessing active="true" count="10" />
            <ExceptionReference name="Analog Exception" />
            <Units>meters</Units>
            <SignificantChange active="true" method="processed" value="1.0" />
            <MaximumChange active="true" method="processed" value="10.0" />
            <Range active="true" method="processed" lower="0.0" upper="100.0" />
            <InitialValue>1.234</InitialValue>
      </AnalogMeasurement>

</ConfigurationDatabase>
```

“Sample Analog 1” shows the simplest analog measurement definition. All properties take on their default values. “Sample Analog 2” shows an analog measurement with most of the valid attributes/elements defined including those inherited from descriptor and measurement. Elements/attributes not explained previously are defined below.

- *ExceptionReference*: Exceptions can be defined in-line or as a reference. In this case, the exception is defined as a reference. See Section 3.4.5 for more information on exceptions.
- *PolynomialConversion*: Conversions can be defined in-line or as a reference. In this case, a polynomial conversion is defined in-line. The polynomial conversion is actually

considered a processing module in CCTK. See Section 3.4.6 for more information on processing modules and processing chains.

- *Units*: The units associated with the measurement, this is an informational field only.
- *InitialValue*: The initial value of measurement can be set with this tag.

3.4.4.3 Defining Discrete Measurements

Discrete measurements can only have one of two values, a high state and a low state. CCTK allows you to define the names of these states and the values they represent. The *DiscreteMeasurement* element is used to declare a discrete measurement in the CCTK configuration database.

Since discrete measurements are a type of measurement and thus a type of descriptor, all of the elements/attributes associated with measurements and descriptors apply to discretes. Table 3-17 lists the elements and attributes associated with a discrete measurement.

Table 3-17: Discrete measurement elements and attributes

Name	**XML Type**	**Data Type**	**Shared**	**Required**	**Default Value**
ExceptionReference	element	n/a	yes	no	(none)
	Defines a reference to an exception for this measurement. This is exclusive with *DiscreteException*.				
DiscreteException	element	n/a	yes	no	(none)
	Defines a discrete exception to associate with this measurement in-line. This is exclusive with *ExceptionReference*.				
DiscreteLowState	element	n/a	yes	no	low, 0
	Defines the low state text and code values for this discrete measurement.				
DiscreteHighState	element	n/a	yes	no	high, 1
	Defines the high state text and code values for this discrete measurement.				
InitialValue	element	string	yes	no	0
	Defines the initial value associated with this measurement. This is specified as either the low or high state text.				

DiscreteLowState and *DiscreteHighState* share a set of common attributes. These attributes are described in Table 3-18.

Table 3-18: *DiscreteLowState* and *DiscreteHighState* attributes

Name	**XML Type**	**Data Type**	**Shared**	**Required**	**Default Value**
code	element	integer	yes	no	0 for low, 1 for high
	Defines the state code to associate with the low or high state of the discrete measurement.				
text	element	string	yes	no	low for low, high for high
	Defines the state text to associate with the low or high state of the discrete measurement.				

Listing 3-18 provides an example of defining a discrete measurement.

Listing 3-18: Configuration database example, defining discrete measurements

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <DiscreteMeasurement name=”Sample Discrete 1” />

      <DiscreteMeasurement name="Sample Discrete 2" active="true" rawDataSize="1">
            <Description>Description for Sample Discrete 2</Description>
            <StaleProcessing active="true" count="10" />
            <DiscreteException name="Inline Discrete Exception">
                  <Exception characteristic="1" notice="Test Notice">x == 0</Exception>
            </DiscreteException>
            <DiscreteLowState code="0" text="low" />
            <DiscreteHighState code="1" text="high" />
            <InitialValue>high</InitialValue>
      </DiscreteMeasurement>

</ConfigurationDatabase>
```

“Sample Discrete 1” shows the simplest discrete measurement definition. All properties take on their default values. “Sample Discrete 2” shows a discrete measurement with most of the valid attributes/elements explicitly defined including those inherited from descriptor and measurement. Elements/attributes not explained previously are defined below.

- *DiscreteException*: Exceptions can be defined in-line or as a reference. In this case, the exception is defined in-line. See Section 3.4.5 for more information on exceptions.
- *DiscreteLowState*: Defines the low state code and value for the discrete measurement.
- *DiscreteHighState*: Defines the high state code and value for the discrete measurement.
- *InitialValue*: The initial value of measurement can be set with this tag.

3.4.4.4 Defining Signed/Unsigned Integer Measurements

Signed/unsigned measurements are a type of measurement used to store an integer value. Signed/unsigned measurements have the native type of *int*, which, on most systems, is a 32 bit IEEE integer value. The signed version is sign extended while the unsigned version is not. The *UnsignedIntMeasurement* element is used to declare an unsigned integer measurement in the CCTK configuration database. The *SignedIntMeasurement* element is used to declare a signed integer measurement in the CCTK configuration database.

Since signed/unsigned integers are a type of measurement and thus a type of descriptor, all of the elements/attributes associated with measurements and descriptors apply to integers. Table 3-19 lists the elements and attributes associated with signed/unsigned measurements.

Table 3-19: Signed/unsigned measurement elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
ExceptionReference	element	n/a	yes	no	(none)
	Defines a reference to an exception for this measurement. This is exclusive with *SignedIntException* (or *UnsignedIntException*).				
SignedIntException	element	n/a	yes	no	(none)
	Defines a signed integer exception to associate with this measurement in-line. This is exclusive with *ExceptionReference*. This element only applies to signed integer measurements.				
UnsignedIntException	element	n/a	yes	no	(none)
	Defines an unsigned integer exception to associate with this measurement in-line. This is exclusive with *ExceptionReference*. This element only applies to unsigned integer measurements.				
Units	element	string	yes	no	(none)
	The units associated with this measurement.				
SignificantChange	element	n/a	yes	no	(inactive)
	Defines the significant change properties associated with this measurement.				
MaximumChange	element	n/a	yes	no	(inactive)
	Defines the maximum change properties associated with this measurement.				
Range	element	n/a	yes	no	(inactive)
	Defines the range properties associated with this measurement.				
InitialValue	element	integer	yes	no	0
	Defines the initial value associated with this measurement.				

Listing 3-19 provides an example of the basic syntax of a signed/unsigned integers.

Listing 3-19: Configuration database example, defining signed and unsigned measurements

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <UnsignedIntMeasurement name=”Sample Int 1” />

      <SignedIntMeasurement name="Sample Int 2" active="true" rawDataSize="4">
            <Description>Description for Sample Int 2</Description>
            <StaleProcessing active="true" count="10" />
            <SignedIntException name="Inline SignedInt Exception" table="mdt">
                  <Exception characteristic="1">x &lt; 0 || x &gt;= 100</Exception>
            </SignedIntException>
            <Units>meters</Units>
            <SignificantChange active="true" method="processed" value="1" />
            <MaximumChange active="true" method="processed" value="10" />
            <Range active="true" method="processed" lower="0" upper="100" />
            <InitialValue>1</InitialValue>
    </SignedIntMeasurement>

</ConfigurationDatabase>
```

"Sample Int 1" shows the simplest unsigned integer measurement definition. All properties take on their default values. "Sample Int 2" shows a signed integer measurement with most of the valid attributes/elements defined including those inherited from descriptor and measurement. Elements/attributes not explained previously are defined below.

- *ExceptionReference*: Exceptions can be defined in-line or as a reference. In this case, the exception is defined in-line. See Section 3.4.5 for more information on exceptions.
- *Units*: The units associated with the measurement, this is reference only field.
- *InitialValue*: The initial value of measurement can be set with this tag.

3.4.4.5 Defining Byte Array Measurements

Byte array measurements store an array of bytes. CCTK allows you to independently specify the maximum storage space allocated for each byte array measurement. The *ByteArrayMeasurement* element is used to declare a byte array measurement in the CCTK configuration database.

Since byte arrays are a type of measurement and thus a type of descriptor, all of the elements/attributes associated with measurements and descriptors apply to byte arrays.

Table 3-20 lists the elements and attributes associated with a byte array measurement.

Table 3-20: Byte array measurement elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
ExceptionReference	element	n/a	yes	no	(none)
	Defines a reference to an exception for this measurement. This is exclusive with *ByteArrayException*.				
ByteArrayException	element	n/a	yes	no	(none)
	Defines a byte array exception to associate with this measurement in-line. This is exclusive with *ExceptionReference*.				
InitialValue	element	(see below)	yes	no	(see below)
	Defines the initial value associated with this measurement. The initial value is inputted as a series of bytes, space delimited. Standard C integer conversion rules will be followed.				

Listing 3-20 provides an example of the basic syntax of the *ByteArrayMeasurement*.

Listing 3-20: Configuration database example, defining byte array measurements

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <ByteArrayMeasurement name="Sample BA 1" />
      <ByteArrayMeasurement name="Sample BA 2" active="true" table="mdt">
            <Description>Description for Sample BA 2</Description>
            <StaleProcessing active="true" count="10" />
            <ExceptionReference name=”Byte Array Exception” />
            <InitialValue>12 23 34 45 56</InitialValue>
      </ByteArrayMeasurement>

</ConfigurationDatabase>
```

“Sample BA 1” in Listing 3-20 shows the simplest byte array measurement definition. All properties take on their default values. “Sample BA 2” shows a byte array measurement with most of the valid attributes/elements defined including those inherited from descriptor and measurement. Elements/attributes not explained previously are defined below.

- *ExceptionReference*: Exceptions can be defined in-line or as a reference. In this case, the exception is defined as a reference. To define an exception in-line, use the *ByteArrayException* element. See Section 3.4.5 for more information on exceptions.
- *InitialValue*: The initial value of the byte array measurement can be set with this tag. The initial value of a byte array is a list of integer numbers between 0-255 (decimal) separated by a space. Each number will be read in turn and placed into the byte array. Standard C conversion rules are followed when processing the numbers. Values that start with 0 are treated as octal, those starting with 0x are treated as hexadecimal.

Byte arrays use the raw data size field slightly differently than the other measurements. With byte arrays, the raw data size field is the maximum storage space allocated for the byte array. When defining a byte array, raw data size should be set to the maximum length of the data that will be stored in this measurement. It is permissible to store less, but never more.

3.4.4.6 Defining String Measurements

String measurements are a subtype of byte array measurements. The previous discussion on byte array measurements applies, except that the element name changes from *ByteArrayMeasurement* to *StringMeasurement*. The key difference lies in the processing of the initial value. For strings, the initial value is considered to be a string and the ASCII values representing the string are inserted into the byte array. Listing 3-21 shows an example of defining a string measurement.

Listing 3-21: Configuration database example, defining string measurements

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <StringMeasurement name=”Sample String 1” />
      <StringMeasurement name="Sample String 2" active="true" table="mdt">
            <Description>Description for Sample String 2</Description>
            <StaleProcessing active="true" count="10" />
            <ExceptionReference name=”String Exception” />
            <InitialValue>This is a string!</InitialValue>
      </StringMeasurement>
</ConfigurationDatabase>
```

3.4.5 Defining Exceptions

Exceptions allow asynchronous events to be generated and captured by CCTK. Exceptions are a special type of descriptor. There is an exception descriptor analogous to each measurement descriptor (i.e. *AnalogMeasurement*/*AnalogException*, *DiscreteMeasurement*/ *DiscreteException,* etc.). Each measurement has the option of defining no exception, defining an exception in-line, or sharing an exception with other measurements using an exception reference.

When an exception is defined in-line, the appropriate exception element for the measurement type (for example, *UnsignedIntException* for an *UnsignedIntMeasurement*) is embedded within the measurement element. Listing 3-18 shows an example of a *DiscreteException* defined in-line.

An exception can also be defined via a reference. When an exception is defined using a reference, the exception descriptor is defined elsewhere in the database and only a reference to the name is embedded within the measurement element. The element *ExceptionReference* is used to reference an exception from within an element. Listing 3-20 shows an example of an *ExceptionReference* used within a *ByteArrayMeasurement*. Table 3-21 presents the attributes associated with the *ExceptionReference* element.

Table 3-21: *ExceptionReference* attributes

Name	XML Type	Data Type	Shared	Required	Default Value
name	attribute	string	yes	yes	–
	Name of the exception that is being referenced.				

Exceptions are defined in the database. Each exception is composed of one or more exception conditions with a maximum number of conditions based upon the type. An exception condition is defined as a mathematical comparison. The generic form of the mathematical comparison is as follows:

x (condition) value (conjunction) x (condition) value

Where x is the current value of the measurement and value, condition, and conjunction are all user input. The valid conjunctions for CCTK are:

<, >, <=, >=, ==, !=

The valid conditions for CCTK are:

&&, ||

An example of an exception condition is:

x < 10 || *x > 20*

Since some conjunctions/conditions are not appropriate for all measurements, Table 3-22 lists the restrictions placed upon each measurement.

Table 3-22: Exception restrictions based upon measurement type

Measurement Type	Number of Conditions	Restrictions
Analog	8	No restrictions.
Discrete	1	Conjunctions cannot be used with discretes. Only the == and != conditions are valid.
SignedInt	8	No restrictions.
UnsignedInt	8	No restrictions.
ByteArray	1	Only the == and != conditions are valid.
String	1	Only the == and != conditions are valid.

Table 3-23 presents the valid attributes for an exception element.

Table 3-23: Exception attributes

Name	XML Type	Data Type	Shared	Required	Default Value
characteristics	attribute	string	yes	no	0
	An integer number that the user may use however is necessary for their application.				
notice	attribute	string	yes	no	(none)
	A link to a notice descriptor that will be generated any time a change in the exception occurs.				

Listing 3-22 presents a sample configuration file with an analog exception defined externally to a group of analog measurements. The analog exception is referenced from the analog measurements via the group defaults. Groups are covered in more detail in Section 3.4.11.

Listing 3-22: Configuration database exception element example

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <AnalogException name="Sample Analog Exception" active="true" table="mdt">
            <Exception characteristic="1" notice="Cautionary Notice">
                        x &lt; 10.0 || x &gt; 90.0</Exception>
            <Exception characteristic="1" notice="Critical Notice">
                        x &lt; 5.0 || x &gt; 95.0</Exception>
      </AnalogException>

      <AnalogMeasurementGroup>
            <AnalogMeasurementDefaults>
                  <ExceptionReference name=”Sample Analog Exception” />
            </AnalogMeasurementDefaults>
            <AnalogMeasurement name=”Analog 1” />
            <AnalogMeasurement name=”Analog 2” />
            <AnalogMeasurement name=”Analog 3” />
      </AnalogMeasurementGroup>
</ConfigurationDatabase>
```

In XML, remember that when placing ‘<’ and ‘>’ in the XML file, you must use the escape sequences ‘&lt;’ and ‘&gt;’ to generate the proper results. If you are using an XML editor, the editor should handle these escape sequences.

3.4.6 Defining Processing Modules

Processing modules perform data processing within CCTK. A processing module is an object that converts data from one format to another using a fixed algorithm and, possibly, a set of inputs. For example, the polynomial conversion module converts a native integer to an analog value using the coefficients configured in the configuration database. Section 3.4.6.2 provides more information on polynomial conversions.

Processing modules can be associated with measurements (see Section 3.4.4) and with link records (see Section 3.4.8). Processing modules are associated with link records when the conversion relates to the interface. Processing modules are associated with measurements when the conversion applies to the measurement regardless of the interface it is received on.

The following sections detail the processing modules associated with the core CCTK system.

3.4.6.1 Defining Integer Conversion Processing Module

The integer conversion module can be used to normalize an input value of varying size into a 64-bit integer value. The integer conversion exists primarily to convert raw integer values of varying data sizes into a format that can be used as the input value for both the polynomial and piecewise linear conversion modules.

The *IntegerConversion* element defines an integer conversion chain. Table 3-24 shows the attributes associated with the integer conversion processing module.

Table 3-24: *IntegerConversion* attributes

Name	XML Type	Data Type	Shared	Required	Default Value
signed	attribute	true or false	no	no	true
	Indicates if the input integer is a signed or an unsigned value.				

Listing 3-23 shows an example of defining an integer conversion processing module.

Listing 3-23: Configuration database example, defining integer conversions

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <IntegerConversion name=”Int 1” signed=”false” />

</ConfigurationDatabase>
```

3.4.6.2 Defining Polynomial Conversion Processing Module

The polynomial conversion module is used to convert a native integer into a CCTK analog measurement. A polynomial conversion is typically configured following an integer conversion in a processing chain. A polynomial conversion takes the generic form:

$$y = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_9x^9$$

The *PolynomialConversion* element defines a polynomial conversion chain, shown in Table 3-25. Table 3-26 lists the attributes of the *PolynomialCoefficient* element.

Table 3-25: *PolynomialConversion* element

Name	XML Type	Data Type	Shared	Required	Default Value
PolynomialCoefficient	element	n/a	no	yes	n/a
	Defines a polynomial coefficient for the polynomial conversion processing module. Each polynomial coefficient can define up to ten polynomial coefficients for a ninth-order polynomial.				

Table 3-26: *PolynomialCoefficient* attributes

Name	XML Type	Data Type	Shared	Required	Default Value
coefficient	attribute	integer	no	yes	n/a
	This represents the coefficient for this value. The coefficient must be between 0-9.				
value	attribute	real	no	yes	n/a
	The value for this coefficient.				

Listing 3-24 shows an example of defining a polynomial conversion processing module.

Listing 3-24: Configuration database example, defining polynomial conversions

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <PolynomialConversion name=”Poly 1”>
            <Description>Description for Poly 1</Description>
            <PolynomialCoefficient coefficient=”0” value=”2.0”>
            <PolynomialCoefficient coefficient=”1” value=”1.0”>
            <PolynomialCoefficient coefficient=”2” value=”0.5”>
      </PolynomialConversion>

      <PolynomialConversion name=”Poly 2”>
            <Description>Description for Poly 2</Description>
            <PolynomialCoefficient coefficient=”0” value=”121.0”>
            <PolynomialCoefficient coefficient=”5” value=”1.5”>
      </PolynomialConversion>

</ConfigurationDatabase>
```

In the above example, “Poly 1” defines the following polynomial:

$$y = (0.5)x^2 + (1.0)x^1 + (2.0)x^0$$

or more simply:

$$y = 0.5x^2 + x + 2$$

While “Poly 2” defines the following polynomial:

$$y = 1.5x^5 + 121$$

Note that not every coefficient needs to be defined. If a coefficient is not defined, its value is assumed to be zero.

3.4.6.3 Defining Piecewise Linear Conversion Processing Module

The piecewise linear conversion module is used to convert a native integer into a CCTK analog measurement. A piecewise linear conversion is typically configured following an integer conversion in a processing chain. A piecewise linear conversion consists of a series of segments defined by an upper and lower bounds. Each segment has an associated linear conversion. To convert a raw value, each segment is checked to see if the raw value is included within the upper and lower bounds. When a segment is found, the segment’s linear conversion is applied to raw value to obtain the processed value. The element used to define a piecewise linear conversion chain is *PiecewiseLinearConversion*. Table 3-27 shows the element associated with the piecewise linear conversion processing chain.

Table 3-27: *PiecewiseLinearConversion* elements

Name	XML Type	Data Type	Shared	Required	Default Value
PiecewiseLinearSegment	element	n/a	no	yes	n/a
	Defines the segments for the piecewise linear conversion. Each segment defines a single linear conversion to apply to the measurement when the value is within the specified range. A maximum of 21 segments may be defined.				

Each *PiecewiseLinearSegment* element has the attributes defined in Table 3-28.

Table 3-28: *PiecewiseLinearSegment* attributes

Name	XML Type	Data Type	Shared	Required	Default Value
low	attribute	integer	no	yes	n/a
	The lower bounds for this segment.				
high	attribute	integer	no	yes	n/a
	The upper bounds for this segment.				
a0	attribute	real	no	yes	n/a
	The 0th order coefficient associated with this conversion.				
a1	attribute	real	no	yes	n/a
	The 1st order coefficient associated with this conversion.				

Listing 3-25 shows an example of defining a piecewise linear conversion processing module.

Listing 3-25: Configuration database example, defining piecewise linear conversions

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <PiecewiseLinearConversion name=”Piece 1”>
            <Description>Description for Piece 1</Description>
            <PiecewiseLinearSegment low=”0” high=”5” a0=”1.5” a1=”1.0” />
            <PiecewiseLinearSegment low=”5” high=”10” a0=”6.5” a1=”1.0” />
            <PiecewiseLinearSegment low=”10” high=”100” a0=”16.5” a1=”2.0” />
      </PiecewiseLinearConversion>

      <PiecewiseLinearConversion name=”Piece 2”>
            <Description>Description for Piece 2</Description>
            <PiecewiseLinearSegment low=”0” high=”200” a0=”0.0” a1=”1.0” />
            <PiecewiseLinearSegment low=”200” high=”1000” a0=”200.0” a1=”1.1” />
      </PiecewiseLinearConversion>

</ConfigurationDatabase>
```

3.4.6.4 Defining Processing Chain Processing Module

CCTK allows a series of processing modules to be linked together to form a processing chain. A processing chain passes the value from module to the next, each module modifying the value with an appropriate algorithm. The *ProcessingChain* element is used to define a processing chain. Table 3-29 shows the valid elements associated with a processing chain.

Table 3-29: *ProcessingChain* elements

Name	XML Type	Data Type	Shared	Required	Default Value
ProcessingReference	element	n/a	yes	no	(none)
	Defines a reference to a processing module for this chain.				
<any processing element>	element	n/a	yes	no	(none)
	Defines a processing module to associate with this chain in-line. In this case, any valid processing element (such as *PolynomialConversion*) can be nested within a processing chain declaration.				

Listing 3-26 shows an example of defining several *ProcessingChain* elements.

Listing 3-26: Configuration database example, defining piecewise linear conversions

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
```

```
<ConfigurationDatabase>

       <IntegerConversion name=”Integer Conversion”>
              [… details removed …]
       </IntegerConversion>
       <PolynomialConversion name=”Polynomial Conversion”>
              [… details removed …]
       </PolynomialConversion>

       <ProcessingChain name=”Chain 1”>
              <ProcessingReference name=”Integer Conversion” />
              <ProcessingReference name=”Polynomial Conversion” />
       </ProcessingChain>

       <ProcessingChain name=”Chain 2”>
              <IntegerConversion name=”In-line Integer Conversion”>
                     [… details removed …]
              </IntegerConversion>
              <PolynomialConversion name=”In-line Polynomial Conversion”>
                     [… details removed …]
              </PolynomialConversion>
       </ProcessingChain>

       <ProcessingChain name=”Chain 3”>
              <ProcessingReference name=”Integer Conversion” />
              <PiecewiseLinearConversion name=”In-line Piecewise Conversion”>
                     [… details removed …]
              </PiecewiseLinearConversion>
       </ProcessingChain>

</ConfigurationDatabase>
```

The above example defines three processing chains. “Chain 1” is defined entirely with references. “Chain 2” is defined entirely with in-lined modules. “Chain 3” is defined with a mix of references and in-lined modules.

Since processing chains are a type of processing module, it is possible to nest a processing chain within another processing chain. This feature can be used to create complex, shared processing chains.

3.4.6.5 Defining User Defined Conversion Processing Module

The user-defined conversion processing module allows you to customize the data processing algorithms within CCTK. Using this module, you can insert custom conversion functions into the standard measurement conversion flow. The attribute for user-defined conversion processing modules is shown in Table 3-30. An example of user-defined conversions is also provided below.

Table 3-30. UserDefinedConversion attributes

Name	XML Type	Data Type	Shared	Required	Default Value
moduleName	attribute	string	n/a	No	UserDefinedConversion
	Defines the conversion algorithm to be used in processing the data coming into the processing module.				

Listing 3-27. Configuration database example, defining user-defined conversions

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <AnalogMeasurement name="Example Analog">
            <UserDefinedConversion name=”Sine Conversion” moduleName=”Sine” />
      </AnalogMeasurement>

</ConfigurationDatabase>
```

The example in Listing 3-27 identifies a custom processing function named Sine to be used in converting the analog measurement’s value from raw to processed form. The example assumes a processing function corresponding to the name Sine has been created and made available as a dynamic processing module. Reference the “Processing Module Development” section in the *CCTK Developer’s Manual* for more details on creating and instantiating custom processing functions to be used by the user defined conversion processing module.

3.4.7 Defining Notices

A notice is a notification of an event or message within a CCTK project. Notices are defined using the *NoticeDescriptor* element. Since notices are a type of descriptor, they inherit all of the properties associated with a Descriptor (see Section 3.4.3). Table 3-31 lists the valid elements/attributes to use when defining a notice descriptor. Table 3-32 and Table 3-33 list the attributes associated with the *NoticeCompression* and *NoticeDistribution* elements respectively.

Table 3-31: Notice descriptor elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
criticality	attribute	string	yes	no	Informational
	Criticality of the notice, must be Informational, Cautionary, or Critical.				
NoticeCompression	element	n/a	yes	no	–
	Defines the compression characteristics for this notice.				
NoticeDistribution	element	n/a	yes	no	–
	Defines the distribution characteristics for this notice.				

The *criticality* attribute indicates the severity of a notice, and is always tied to a particular notice. The possible values of *criticality* are “Informational”, “Critical”, and “Cautionary”. The default value is “Informational”.

Table 3-32: NoticeCompression attributes

Name	Type	Shared	Required	Default Value
active	attribute	yes	no	true
	Indicates if notice compression is active for this notice.			
minimumMicroseconds	attribute	yes	no	100
	Indicates the minimum number of microseconds, which must pass before CCTK will process the same system notice twice in a row.			

Notice compression is used to limit the flow of a notice through the system by enforcing a minimum time between notices. If notice compression is active for a notice, that notice will not be processed by the system if it is received within *minimumMicroseconds* of the last time that notice was received. The primary purpose of notice compression is to reduce the system load when a single message is repeatedly sent through the system in a short period of time. Notice compression is inactive by default.

Table 3-33: NoticeDistribution attributes

Name	XML Type	Data Type	Shared	Required	Default Value
distributeOverChannel	attribute	true or false	yes	no	true
	Indicates that the notice should be distributed over channels using the CCTK standard distribution methods.				
distributeOverFile	attribute	true or false	yes	no	true
	Indicates that the notice should be placed in the system notice file in the log directory.				
distributeToChannels	attribute	string	yes	no	STDARCH
	Lists the channels that the notice packet should be written to. This is usually only set to the default of STDARCH.				

Notice distribution controls which processes or files receive a notice after it is generated. If *distributeOverChannel* is active, the notice will be sent to any process that has registered to receive it. This typically includes the archive subsystem and any user process that may have registered for notices. Therefore, to have a notice archived, *distributeOverChannel* must be active and *distributeToChannels* set to STDARCH. This is the default. If *distributeOverFile* is active, the notice will be added to the global system notice file, which contains a list of all notices generated during the execution of the current project. This file is used by the *SysMsgGui* to obtain a list of system messages. So, if this notice is intended to be displayed in the *SysMsgGui* application, *distributeOverFile* must be active.

Listing 3-28 presents an example of defining notices in the configuration database.

Listing 3-28: Configuration database example, defining notice descriptors

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <NoticeDescriptor name=”Sample Notice 1” />

      <NoticeDescriptor name="Sample Notice 2" table="ndt" criticality="Informational">
            <Description>Description for Sample Notice 2</Description>
            <NoticeCompression active="true" minimumMicroseconds="100" />
            <NoticeDistribution distributeOverChannel="yes" distributeOverFile="yes" />
      </NoticeDescriptor>

</ConfigurationDatabase>
```

“Sample Notice 1” shows the minimum required fields for defining a notice descriptor. Only the name attribute is necessary. “Sample Notice 2” shows all of the fields that are valid for a notice descriptor. The fields not described in the descriptor section are discussed below.

3.4.8 Defining Link Records

Link records are used to describe a block of information. Link records are used in both command definition and packet decommutation definition. Since link records are not a descriptor, they cannot appear alone in the configuration database. They must be embedded in another descriptor element. This section describes the basics of defining link records. They will be further expanded upon in the packet decommutation sections.

There are three types of link records defined in the CCTK configuration database:

- *MeasurementLinkRecord*: Used in *PacketDecomDescriptor* to link a measurement to a block of data.
- *InlineCommandParameterLinkRecord*: Used in the commanding descriptors to specify a command parameter with its data in-line with the link record.
- *InputCommandParameterLinkRecord*: Used in the commanding descriptors to specify a command parameter that is passed in as part of the command.
- *PredefinedCommandParameterLinkRecord*: used in the command descriptors to specify a command parameter that is predefined in the database.

Each link record is used to describe the position of a piece of data within a block of bytes. Therefore, all link records share a common set of attributes that are used to locate the data within the block of bytes. Each of the link records defined above support the attributes listed in Table 3-34.

Table 3-34: Shared link record attributes

Name	XML Type	Data Type	Shared	Required	Default Value
active	attribute	true or false	n/a	no	true
	Indicates if this link record is active. If a link record is inactive, it will be ignored during processing.				
startWord	attribute	integer	n/a	yes	n/a
	The starting word associated with this link record.				
startBit	attribute	integer	n/a	yes	n/a
	The starting bit associated with this link record.				
bitLength	attribute	integer	n/a	yes	n/a
	The bit length associated with this link record.				

Once the data within the block of bytes is located, the link record is used to link that block with some value or measurement. The *InlineCommandParameterLinkRecord* links the block of data to a value that is stored with the link record. The other link records link the block of data to a CCTK measurement. The following tables show the specific fields associated with each link record.

InlineCommandParameterLinkRecord supports the additional attribute listed in Table 3-35.

Table 3-35: *InlineCommandParameterLinkRecord* attribute

Name	XML Type	Data Type	Shared	Required	Default Value
value	attribute	integer	–	yes	–
	The value associated with this link record.				

MeasurementLinkRecord, *InputCommandParameterLinkRecord*, and *PredefinedCommandParameterLinkRecord* support the additional elements listed in Table 3-36.

Table 3-36: MeasurementLinkRecord, InputCommandParameterLinkRecord, and PredefinedCommandParameterLinkRecord elements

Name	XML Type	Data Type	Shared	Required	Default Value
MeasurementReference	element	n/a	n/a	yes	n/a
	A reference to a measurement descriptor.				
{any}Measurement	element	n/a	n/a	yes	n/a
	A measurement descriptor defined in-line with the link record. This can be either AnalogMeasurement, DiscreteMeasurement, UnsignedIntegerMeasurement, SignedIntegerMeasurement, etc …				

Only one measurement may be referenced from these link records. The measurement reference can be in the form of a *MeasurementReference* element, with the appropriate name specified. The measurement reference can also be in the form of an in-line measurement declaration where the measurement is declared in-line with the link record. In this case, any valid measurement element may be placed in-line.

Table 3-37 shows the valid attributes for the *MeasurementReference* element.

Table 3-37: *MeasurementReference* elements

Name	XML Type	Data Type	Shared	Required	Default Value
name	attribute	string	n/a	yes	n/a
	The name of the measurement being referenced.				

3.4.9 Defining Packet Decommutation Records

The *PktDcom* process uses information stored in packet decommutation records to break blocks of data apart into individual measurements. Many different types of external interfaces generate blocks of data in a consistent, defined format. For these interfaces, *PktDcom* can be used to break the block of data into individual measurements. PCM is one example of an external interface that uses *PktDcom* to generate measurements from blocks of data.

PktDcom records are defined in the database using the *PacketDcomDescriptor* element. Table 3-38 lists the valid elements for a *PacketDcomDescriptor*.

Table 3-38: *PktDcom* descriptor elements

Name	XML Type	Data Type	Shared	Required	Default Value
RawMeasurementLinkRecord	element	n/a	yes	no	(none)
	Defines the raw measurement link record for this decommutation descriptor. The raw measurement is typically the entire block of data, but it does not have to be. This special record is used when an interface allows raw dump to be enabled/disabled.				
MeasurementLinkRecord	element	n/a	yes	no	(none)
	This element lists the measurement link records associated with this decommutation record. Typically, multiple *MeasurementLinkRecords* are defined for each *PacketDcomDescriptor*.				

Listing 3-29 shows an example of a packet decom descriptor.

Listing 3-29: Configuration database example, defining packet decommutation descriptors

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <AnalogMeasurement name=”Measurement Reference 1” />
      <AnalogMeasurement name=”Measurement Reference 2” />

      <PacketDcomDescriptor name="Packet Decom">
            <Description>Packet Decom Description</Description>
            <RawMeasurementLinkRecord startWord="0" startBit="0" bitLength="32">
                  <MeasurementReference name="Measurement Reference 1"/>
            </RawMeasurementLinkRecord>
            <MeasurementLinkRecord startWord="0" startBit="0" bitLength="8">
                  <AnalogMeasurement name="Inline Analog" table="mdt"/>
            </MeasurementLinkRecord>
            <MeasurementLinkRecord startWord="0" startBit="8" bitLength="8">
                  <MeasurementReference name="Measurement Reference 2"/>
            </MeasurementLinkRecord>
            <MeasurementLinkRecord startWord="1" startBit="0" bitLength="1">
                  <DiscreteMeasurement name="Inline Discrete ">
                        <Description>Test Description</Description>
                  </DiscreteMeasurement>
            </MeasurementLinkRecord>
      </PacketDcomDescriptor>

</ConfigurationDatabase>
```

Note that in many cases, the *PktDcomDescriptor* element is actually defined in-line in an interface descriptor.

3.4.10 Defining Interfaces

CCTK always allows at least two interface types to be defined:

- Archive: The archive interface provides configuration and control of the archive subsystem within CCTK.
- Time Control: The time control interface provides configuration and control of the GMT and CDT timing subsystems within CCTK.

These interfaces are discussed in detail in the following sections. In addition, the optional peer interface is detailed in this section.

CCTK can also be customized to support additional external data interfaces. Since each interface has a unique definition, they are described in detail in the documentation included with the particular interface.

Commands, ports, buses, status measurements, and notices are all typically defined automatically by interface creation.

3.4.10.1 Defining an ARS Interface

If CCTK has an archive subsystem, an archive interface must be defined. The archive interface is the point of configuration for the archive subsystem processes. Details on the execution of the archive subsystem processes can be found in Section 3.5.2. Both the database configuration and the process execution must be present in a CCTK project for the archive subsystem to work correctly.

Only one archive subsystem interface may be defined in a CCTK project and it must be defined as shown below in Listing 3-30 for correct operation.

Listing 3-30: Configuration database example, defining the archive interface

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <ArsInterface name="ARS_port_descriptor">
            <ArsStartCommand name="Start_ARS" />
            <ArsStopCommand name="Stop_ARS" />
            <ArsCloseArchiveCommand name="ARS_Close_Archive" />
            <ArsStatusMeasurement name="ARS_Status" table="mdt"/>
      </ArsInterface>

</ConfigurationDatabase>
```

3.4.10.2 Defining a Time Control Interface

If CCTK needs to manage countdown time, a time control interface must be defined. The time control interface is the point of configuration for the time control process. Details on the executing the time control process can be found in Section 3.5.6. Both the database configuration and the process execution must be present in a CCTK project for time control to work correctly.

Only one time control interface may be defined in a CCTK project and it must be defined as shown below in Listing 3-31 for correct operation.

Listing 3-31: Configuration database example, defining the time control interface

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <TimeControlInterface name="CDTime_port">
            <TimeControlCancelCdtCommand name="Cancel_CDTime"/>
            <TimeControlSetCdtCommand name="Set_CDTime"/>
            <TimeControlStartCdtCommand name="Start_CDTime"/>
            <TimeControlStopCdtCommand name="Stop_CDTime"/>
            <TimeControlCdtStatusMeasurement name="CDTime_Status"/>
            <TimeControlCdtTextMeasurement name="CDTime_Time_Text"/>
            <TimeControlCdtValueMeasurement name="CDTime_Time_Value"/>
            <TimeControlGmtTextMeasurement name="GMTime_Time_Text"/>
            <TimeControlGmtValueMeasurement name="GMTime_Time_Value"/>
      </TimeControlInterface>

</ConfigurationDatabase>
```

3.4.10.3 Defining a Peer Sender Interface

PeerSender allows CCTK to transmit data from the sending peer to the receiving peer via a network connection. *PeerSender* can be configured to support both connection and connectionless sockets. Broadcast and multicast support is also provided when a connectionless socket is configured. This section details how to optionally configure *PeerSender* through the configuration database. If *PeerSender* is not configured through the configuration database, it must be configured via the command line arguments. Section 3.5.8.1 provides additional information on the *PeerSender* process.

The *PeerSenderInterface* element controls the configuration information associated with a *PeerSender* process. Since *PeerSenderInterface* is a type of descriptor, therefore all of the elements/attributes associated with descriptors apply to it. Table 3-39 lists the elements unique to the *PeerSenderInterface*.

Table 3-39: *PeerSenderInterface* elements

Name	XML Type	Data Type	Shared	Required	Default Value
SocketPort	element	n/a	n/a	yes	(none)
	This element defines the properties associated with the socket. See Table 3-40 for the elements and attributes associated with this element. This element is required.				
PeerSenderStartDataTransmissionCommand	element	n/a	n/a	no	(none)
	This element allows the user to define a custom start data transmission command. The user may override the name of the command as well as any other properties of the command. This element is a type of *SystemCommand*. Under normal circumstances, this should not need to be overridden.				
PeerSenderStopDataTransmissionCommand	element	n/a	n/a	no	(none)
	This element allows the user to define a custom stop data transmission command. The user may override the name of the command as well as any other properties of the command. This element is a type of *SystemCommand*. Under normal circumstances, this should not need to be overridden.				
PeerSenderDataTransmissionStatusMeasurement	element	n/a	n/a	no	(none)
	This element allows the user to define a custom data transmission status measurement. The user may override the name of the status measurement as well as any other properties of the measurement. This element is a type of *DiscreteMeasurement*. Under normal circumstances, this should not need to be overridden.				

As with most interfaces in CCTK, the configuration database allows the user to override the default commands and measurements. This allows the user to change the properties of the command/measurement within the limits of the type that is required (i.e. *PeerSenderDataTransmissionStatusMeasurement* is a discrete measurement so only the properties associated with a discrete may be modified). Unless you wish to change the name of a command/measurement, it is recommended that you do not override the default settings.

Each *PeerSender* is associated with a socket. The socket is configured through the *SocketPort* element. Table 3-40 lists the attributes associated with a *SocketPort*.

Table 3-40: *PeerSenderInterface SocketPort* attributes

Name	XML Type	Data Type	Shared	Required	Default Value

Name	XML Type	Data Type	Shared	Required	Default Value
mode	attribute	n/a	n/a	yes	(none)
	This attribute can take one of five values: pointtopoint, connectionless, broadcast, multicast, or server. This attribute indicates the mode in which the socket should be opened. This attribute is required.				
localAddress	attribute	n/a	n/a	(see notes)	(see notes)
	This attribute specifies the local address for the socket. The local address is optional for all modes. If not specified, it defaults to any available address. The local address must always be a valid address for the local host. *localAddress* may be specified as a textual string or as an IP address in quad notation (xxx.xxx.xxx.xxx). If a string is specified, a standard lookup will be performed.				
localPort	attribute	n/a	n/a	(see notes)	(see notes)
	This attribute specifies the local port for the socket. The local port is required for point-to-point and server and optional for the other modes. If not specified, local port defaults to any port. The local port cannot be below 1024 unless you are running as root. *localPort* may be specified as a textual string or as a port number. If a string is specified, a standard services lookup will be performed.				
remoteAddress	attribute	n/a	n/a	(see notes)	(see notes)
	This attribute specifies the remote address for the socket. The remote address is ignored when operating in point-to-ponit or server mode. The remote address is required for connectionless and multicast. It is optional for broadcast. If not specified in broadcast mode, it defaults to "255.255.255.255". *remoteAddress* may be specified as a textual string or as an IP address in quad notation (xxx.xxx.xxx.xxx). If a string is specified, a standard lookup will be performed.				
remotePort	attribute	n/a	n/a	(see notes)	(see notes)
	This attribute specifies the remote port for the socket. The remote port is required for connectionless, broadcast, and multicast. It is ignored for the other two modes. The remote port cannot be below 1024 unless you are running as root. *remotePort* may be specified as a textual string or as a port number. If a string is specified, a standard services lookup will be performed.				

PeerSender can operate in one of five different modes. Each mode represents a different socket configuration, and in some instances, different behavior. Each mode has slightly different configuration needs for addresses/ports. Table 3-41 summarizes the optional/required ports/addresses for a *PeerSenderInterface* based upon mode. For more information on the individual modes, please see Section 3.5.8.1.

Table 3-41: *PeerSender* Port/Address Requirements

Mode	Local Address	Local Port	Remote Address	Remote Port
Server	optional	required	ignored	ignored
Point-to-Point	optional	required	ignored	ignored
Connectionless	optional	optional	required	required
Broadcast	optional	optional	optional	required
Multicast	optional	optional	required	required

Listing 3-32 presents an example of defining peer sender interfaces in the configuration database.

Listing 3-32: Configuration database example, defining the peer sender interface

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

      <PeerSenderInterface name="Sender #1">
            <SocketPort mode=”pointtopoint” localPort=”10000” />
      </PeerSenderInterface>

      <PeerSenderInterface name=”Sender #2”>
            <SocketPort mode=”connectionless” remoteAddress=”192.168.1.1” 
remotePort=”20000” />
      </PeerSenderInterface>

      <PeerSenderInterface name=”Sender #3”>
            <SocketPort mode=”broadcast” localAddress=”192.168.1.1” localPort=”20000” 
remoteAddress=”192.168.255.255” remotePort=”20001” />
            <PeerSenderStartDataTransmissionCommand name=”Start Sender #3” />
            <PeerSenderStopDataTransmissionCommand name=”Stop Sender #3” />
            <PeerSendrDataTransmissionStatusMeasurement name=”Sender #3 Status” />
      </PeerSenderInterface>
</ConfigurationDatabase>
```

“Sender #1” defines a *PeerSenderInterface* operating in point to point mode. The local port will be bound to port 10000. The local address will default to any available address (thus a connection could be received on any Ethernet interface for a multi-homed host). In this mode, *PeerSender* will accept a single connection from a remote receiver. Once the connection is established, data transmission will begin.

“Sender #2” defines a *PeerSenderInteface* operating in connectionless mode. The remote address and remote port data will be sent to is “192.168.1.1” and “20000” respectively. Any available local address/port will be used for local socket. In this mode, data transmission will begin immediately.

“Sender #3” defines a *PeerSenderInterface* operating in broadcast mode. The local address is “192.168.1.1”. This must be valid Ethernet address for the host. The local port is “20000”. When broadcasting the data, it will be broadcast to the address “192.168.255.255” and to the port “20001”. In addition, the names of the start data transmission command, stop data transmission command, and data transmission status measurement were changed from their defaults. In this mode, data transmission will begin when the data transmission enabled command is received.

3.4.10.4 Defining a Peer Receiver Interface

PeerReceiver allows CCTK to receive data from a sending peer via a network connection. *PeerReceiver* can be configured to support both connection and connectionless sockets Connectionless sockets have the capability to broadcast and multicast data. This section details how to optionally configure *PeerReceiver* through the configuration database. If *PeerReceiver* is not configured through the configuration database, it must be configured via the command line arguments. Section 3.5.8.2 provides additional information on the *PeerReceiver* process.

The *PeerReceiverInterface* element contains the configuration information associated with a *PeerReceiver* process. Since *PeerReceiverInterface* is a type of descriptor, therefore all of the elements/attributes associated with descriptors apply to it. Table 3-42 lists the elements unique to the *PeerReceiverInterface*.

Table 3-42: *PeerReceiverInterface* elements

Name	XML Type	Data Type	Shared	Required	Default Value
SocketPort	element	n/a	n/a	yes	(none)
	This element defines the properties associated with the socket. See Table 3-43 for the elements and attributes associated with this element. This element is required.				
PeerReceiverStartDataAcquisitionCommand	element	n/a	n/a	no	(none)
	This element allows the user to define a custom start data acquisition command. The user may override the name of the command as well as any other properties of the command. This element is a type of *SystemCommand*. Under normal circumstances, this should not need to be overridden.				
PeerReceiverStopDataAcquisitionCommand	element	n/a	n/a	no	(none)
	This element allows the user to define a custom stop data acquisition command. The user may override the name of the command as well as any other properties of the command. This element is a type of *SystemCommand*. Under normal circumstances, this should not need to be overridden.				
PeerReceiverDataAcquisitionStatusMeasurement	element	n/a	n/a	no	(none)
	This element allows the user to define a custom data acquisition status measurement. The user may override the name of the status measurement as well as any other properties of the measurement. This element is a type of *DiscreteMeasurement*. Under normal circumstances, this should not need to be overridden.				
PeerReceiverStartDataProcessingCommand	element	n/a	n/a	no	(none)
	This element allows the user to define a custom start data processing command. The user may override the name of the command as well as any other properties of the command. This element is a type of *SystemCommand*. Under normal circumstances, this should not need to be overridden.				
PeerReceiverStopDataProcessingCommand	element	n/a	n/a	no	(none)
	This element allows the user to define a custom stop data processing command. The user may override the name of the command as well as any other properties of the command. This element is a type of *SystemCommand*. Under normal circumstances, this should not need to be overridden.				
PeerReceiverDataProcessingStatusMeasurement	element	n/a	n/a	no	(none)
	This element allows the user to define a custom data processing status measurement. The user may override the name of the status measurement as well as any other properties of the measurement. This element is a type of *DiscreteMeasurement*. Under normal circumstances, this should not need to be overridden.				
MeasurementsToReceive	element	n/a	n/a	(see below)	(none)
	This element allows the user to define a mapping between measurements on the sender system and measurements on the remote system. It lists the measurements the user is to receive.				

As with most interfaces in CCTK, the configuration database allows the user to override the default commands and measurements. This allows the user to change the properties of the command/measurement within the limits of the type that is required (i.e. *PeerReceiverDataAcquisitionStatusMeasurement* is a discrete measurement so only the properties associated with a discrete may be modified). Unless you wish to change the name of a command/measurement, it is recommended that you do not override the default settings.

Table 3-43 lists the attributes associated with a *SocketPort*.

Table 3-43: *PeerReceiverInterface SocketPort* attributes

Name	XML Type	Data Type	Shared	Required	Default Value
mode	attribute	string	n/a	yes	(none)
	This attribute can take one of four values: point-to-point, connectionless, broadcast, or multicast. This attribute indicates the mode in which the socket should be opened. This attribute is required.				
localAddress	attribute	string	n/a	(see notes)	(see notes)
	This attribute specifies the local address for the socket. The local address is optional for connectionless, broadcast, and multicast. It is ignored in point-to-point. If not specified, it defaults to any available address. The local address must always be a valid address for the local host. *localAddress* may be specified as a textual string or as an IP address in quad notation (xxx.xxx.xxx.xxx). If a string is specified, a standard lookup will be performed.				
localPort	attribute	string	n/a	(see notes)	(see notes)
	This attribute specifies the local port for the socket. The local port is required for connectionless, broadcast, and multicast. It is ignored for point-to-point. If not specified, local port defaults to any port. The local port cannot be below 1024 unless you are running as root. *localPort* may be specified as a textual string or as a port number. If a string is specified, a standard services lookup will be performed.				
remoteAddress	attribute	string	n/a	(see notes)	(see notes)
	This attribute specifies the remote address for the socket. The remote address is optional for connectionless and multicast. It is required for point-to-point. *remoteAddress* may be specified as a textual string or as an IP address in quad notation (xxx.xxx.xxx.xxx). If a string is specified, a standard lookup will be performed.				
remotePort	attribute	string	n/a	(see notes)	(see notes)
	This attribute specifies the remote port for the socket. The remote port is optional for connectionless, broadcast, and multicast. It is required for point-to-point. The remote port cannot be below 1024 unless you are running as root. *remotePort* may be specified as a textual string or as a port number. If a string is specified, a standard services lookup will be performed.				

PeerReceiver can operate in one of four different modes. Each mode represents a different socket configuration, and in some instances, different behavior. Each mode has slightly different configuration needs for addresses/ports. Table 3-44 summarizes the optional/required ports/addresses for a *PeerReceiverInterface* based upon mode. For more information on the individual modes, please see Section 3.5.8.2.

Table 3-44: *PeerReceiver* Port/Address Requirements

Mode	Local Address	Local Port	Remote Address	Remote Port
Point-to-Point	ignored	ignored	required	required
Connectionless	optional	required	optional	optional

Mode	Local Address	Local Port	Remote Address	Remote Port
Broadcast	optional	required	optional	optional
Multicast	optional	required	optional	optional

Table 3-45 lists the element associated with a *MeasurementsToReceive* element. The sole purpose of this element is to provide for a list of measurements that define how to map measurements between the sender and the receiver.

Table 3-45: *MeasurementsToReceive* elements

Name	XML Type	Data Type	Shared	Required	Default Value
MapMeasurement	element	n/a	n/a	yes	(none)
	This element provides data for mapping a measurement on the local system to a measurement on the remote system. Multiple *MapMeasurements* may be specified within a single *MeasurementsToRecieve*.				

Table 3-46 lists the elements and attributes associated with *MapMeasurement*.

Table 3-46: *MapMeasurement* elements and attributes

Name	XML Type	Data Type	Shared	Required	Default Value
remoteName	attribute	string	n/a	yes	(none)
	The name of the measurement on the remote system.				
active	attribute	n/a	n/a	no	true
	Indicates if this mapping should be active or inactive. If active is set to false, this mapping will not be processed.				
MeasurementReference	element	n/a	n/a	yes	n/a
	A reference to a measurement descriptor. Only one of *{any}Measurement* or *MeasurementReference* may be defined.				
{any}Measurement	element	n/a	n/a	yes	n/a
	A measurement descriptor defined in-line with the link record. This can be either *AnalogMeasurement*, *DiscreteMeasurement*, *UnsignedIntegerMeasurement*, *SignedIntegerMeasurement*, etc … Only one of *{any}Measurement* or *MeasurementReference* may be defined.				

Each *MapMeasurement* element defines the mapping between a measurement received from the remote sender and a measurement on the local receiver. This provides a simple way for the user to associate measurements on the remote system with measurements on the local system. This is one method used by *PeerReceiver* to map measurements between the peers. Section 3.5.8.2 describes additional methods thus, it may only be necessary to specify this section if the names are different between the sending and receiving peers.

Listing 3-33 presents an example of defining peer receiver interfaces in the configuration database.

Listing 3-33: Configuration database example, defining the peer receiver interface

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase>

       <PeerReceiverInterface name="Receiver #1">
              <SocketPort mode=”connectionless” localPort=”10000” />
       </PeerReceiverInterface>

       <PeerReceiverInterface name=”Receiver #2”>
              <SocketPort mode=”pointtopoint” remoteAddress=”192.168.1.1”
remotePort=”20000” />
              <MeasurementsToReceive>
                     <MapMeasurement remoteName=”Remote Meas #1”>
                            <MeasurementReference name=”Local Meas #1”/>
                     </MapMeasurement>
                     <MapMeasurement remoteName=”Remote Meas #2”>
                            <AnalogMeasurement name=”Local Meas #2” />
                     </MapMeasurement>
                     <MapMeasurement remoteName=”Remote Meas #3”>
                            <DiscreteMeasurement name=”Local Meas #3” />
                     </MapMeasurement>
              </MeasurementsToReceive>
       </PeerReceiverInterface>

       <PeerReceiverInterface name=”Sender #3”>
              <SocketPort mode=”broadcast” localAddress=”192.168.1.1” localPort=”20000”
remoteAddress=”192.168.255.255” remotePort=”20001” />
              <PeerReceiverStartDataAcquisitionCommand name=”Start Acq Receiver #3” />
              <PeerReceiverStopDataAcquisitionCommand name=”Stop Acq Receiver #3” />
              <PeerReceiverDataAcquisitionStatusMeasurement name=”Receiver #3 Acq Status”
/>
              <PeerReceiverStartDataProcessingCommand name=”Start Proc Receiver #3” />
              <PeerReceiverStopDataProcessingCommand name=”Stop Proc Receiver #3” />
              <PeerReceiverDataProcessingStatusMeasurement name=”Receiver #3 Proc Status”
/>
       </PeerSenderInterface>
</ConfigurationDatabase>
```

“Receiver #1” defines a *PeerReceiverInterface* operating in connectionless mode. The local port will be bound to port “10000”. The local address will default to any available address (thus a connection could be received on any Ethernet interface for a multihomed host).

“Receiver #2” defines a *PeerReceiverInteface* operating in point-to-point mode. A connection attempt will be made to the remote address “192.168.1.1” and the remote port “20000”. Any available local address/port will be used for local socket. In addition, the measurements listed will be mapped. “Remote Meas #1” will be mapped to “Local Meas #1”, “Remote Meas #2” will be mapped to “Local Meas #2”, and “Remote Meas #3” will be mapped to “Local Meas

#3”. The user must ensure that the data types for the different measurements match. If they do not, conversions must be provided to convert the measurements between data types.

“Receiver #3” defines a *PeerReceiverInterface* operating in broadcast mode. The local address is “192.168.1.1”. This must be valid Ethernet address for the host. The local port is “20000”. When receiving the data, the data must be addressed to address “192.168.255.255” and to port “20001”. In addition, the names of all commands and status measurements were changed from their original values.

3.4.11 Using Groups

One of the most powerful features of the CCTK configuration database is the ability to group items together that share common properties. When a group is created, all items of the group inherit the default properties associated with that group. It is possible for individual items within the group to override the default properties. Groups can be nested within one another to form a type of inheritance tree. Listing 3-34 presents the basic grouping syntax. Listing 3-35 shows an example.

Listing 3-34: Configuration database general grouping syntax

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase shlibs=”[… dynamic libraries …]”>

       <ParentGroup>
              <ParentDefaults [… default attributes associated with parent …]>
                     [… default elements associated with parent …]
              </ParentDefaults>
              <ChildGroup>
                     <ChildDefaults [… default attributes associated with child …]>
                            [… default elements associated with child …]
                     </ChildDefaults>
                     <Child [… any attribute associated with child …]>
                            [… child elements …]
                     </Child>
                     [… more children …]>
              </ChildGroup>
              [… more parent, parent group, children or child groups …]
       </ParentGroup>

</ConfigurationDatabase>
```

In the above listing, the ParentGroup element has a set of defaults defined by the ParentDefaults element. Although only one child is shown within the ParentGroup element, any number of elements could be listed here. These could be ChildGroup or Child elements. In addition, since the configuration database allows recursive groups, both Parent and

ParentGroup elements could appear within the top level ParentGroup. All elements within the ParentGroup inherit the defaults of the group. Within the listed ChildGroup, another set of defaults is shown. These defaults apply to the all elements within the ChildGroup. Again, only one Child is shown here, but any number of Child and ChildGroup elements could be placed here.

Appending group onto the name of an element discussed above creates a group. For example, a group of analog measurements can be defined using the *AnalogMeasurementGroup* element. Table 3-47 lists the previously discussed elements that may be grouped.

Table 3-47: Configuration database elements that may be grouped

Type	**Element Name**	**Element Group Name**
descriptors	–	*DescriptorGroup*
notice descriptors	*NoticeDescriptor*	*NoticeDescriptorGroup*
measurements	–	*MeasurementGroup*
analog measurement	*AnalogMeasurement*	*AnalogMeasurementGroup*
discrete measurement	*DiscreteMeasurement*	*DiscreteMeasurementGroup*
signed integer measurement	*SignedIntMeasurement*	*SignedIntMeasurementGroup*
unsigned integer measurement	*UnsignedIntMeasurement*	*UnsignedIntMeasurementGroup*
byte array measurement	*ByteArrayMeasurement*	*ByteArrayMeasurementGroup*
string measurement	*StringMeasurement*	*StringMeasurementGroup*

Listing 3-35 provides a real example of a group using measurements.

Listing 3-35: Configuration database grouping example

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd">
<ConfigurationDatabase shlibs=”[… dynamic libraries …]”>

      <MeasurementGroup>
            <MeasurementDefaults />
            <MeasurementGroup rawDataSize=”4”>
                  <MeasurementDefaults />
                  <AnalogMeasurement name=”Analog 1”>
                        <Description>Description for Analog 1</Description>
                  </AnalogMeasurement>
                  <AnalogMeasurement name=”Analog 2”>
                        <Description>Description for Analog 2</Description>
                  </AnalogMeasurement>
            </MeasurementGroup>
            <MeasurementGroup>
                  <MeasurementDefaults />
                  <DiscreteMeasurementGroup>
                        <DiscreteMeasurementDefaults>
                              <DiscreteLowState code=”10” text=”Off” />
                              <DiscreteHighState code=”20” text=”On” />
                        </DiscreteMeasurementDefaults>
```

```
                                        <DiscreteMeasurement name=”Discrete 1” />
                                        <DiscreteMeasurement name=”Discrete 2” />
                                        <DiscreteMeasurement name=”Discrete 3”>
                                                  <DiscreteLowState code=”0” text=”Off” />
                                        </DiscreteMeasurement>
                              </DiscreteMeasurementGroup>
                    </MeasurementGroup>
          </MeasurementGroup>

</ConfigurationDatabase>
```

In the above example, the *DescriptorGroup* contains two *MeasurementGroups.* The first *MeasurementGroup* contains two *AnalogMeasurements*, which inherit the *rawDataSize* attribute from *MeasurementGroup*. Each *AnalogMeasurement* defines its own name and description. The second *MeasurementGroup* contains a single *DiscreteMeasurementGroup*. The *DiscreteMeasurementGroup* contains three discretes, which inherit the high/low state values/codes from the group. In addition, one discrete overrides the default values and defines its own low state value/code.

Note that a hierarchy of types is present throughout the grouping. A *DiscreteMeasurement* is a type of Measurement which is a type of Descriptor thus *DiscreteMeasurement* can be placed inside a *DiscreteMeasurementGroup*, a *MeasurementGroup*, or a *DescriptorGroup*.

Another important point to note on groups is that not all elements and attributes associated with a type may be capable of being defined in a group. For example, the name attribute, which is associated with every descriptor, cannot appear as a default attribute. It would make no sense to share a name between multiple descriptors. As each table/descriptor was presented, the different attributes and sub-elements are discussed and any that can not be shared across groups are noted. The tables listing the elements and attributes for the descriptors in the above sections all contain a “shared” column, which indicates if the element/attribute may be shared across groups.

A final note, many attributes and elements have default values. If the element/attribute is not specified, the default value will be used.

3.4.12 Custom Database Components

You can extend the CCTK configuration database by adding new tables and descriptors. When this occurs, you must define a new document type, a new dtd, and create a shared library with the code needed to load the new tables/descriptors. The *shlibs* attribute is used to specify the shared libraries needed to load custom descriptors/tables. For more information on extending the CCTK configuration database, reference the *CCTK Developer’s Manual* or contact CCT.

3.4.13 Standard System Notices

For CCTK to operate properly, a standard set of system notices must be defined in the database. These system notices are required by the core processes to properly report errors in the system. The standard system notices are included in the database by using the XML entity reference to include the file containing the standard system notices. The files name is $*CCT_HOME*/include/dtd/SystemNotice.xml.

A partial configuration database file that shows how to perform the entity inclusion is shown in Listing 3-36.

Listing 3-36: Standard system notice file inclusion

```
<?xml version="1.0"?>
<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd" [
    <!ENTITY SystemNotices SYSTEM "SysNotice.xml">
    ]>

<ConfigurationDatabase>
      [… be sure that the ndt table is defined prior to inclusion …]
      &SystemNotices;
</ConfigurationDatabase>
```

3.4.14 Example Configuration Database

Several example configuration database files are present in the CCT examples directory ($*CCT_HOME*/examples). See the README file in that directory for more details. In addition, the sample projects also contain sample databases at $*CCT_HOME*/projects.

3.5 Configuration of System Tasks

Each CCTK project is composed of a set of processes that are responsible for different tasks in the system. Some processes perform archiving tasks (*ArchiveControl* and *TamArs*) while others perform data processing tasks (*DataProc* and *PktDcom*). By configuring which processes run for a specific project (or modes within a project), the project administrator can control the operation of CCTK. This section discusses the tasks that must be performed for correct operation of CCTK and the processes that perform these tasks.

3.5.1 Messaging

Messaging is an essential service of CCTK. System notices are the mechanism by which messaging is implemented in CCTK. System notices can be added to the configuration database (see Section 3.4.7). The developer's API provides a simple call to generate a message (see the *CCTK Developer's Guide*). Messages can be viewed from the system message GUI (see the *CCTK User's Guide*). Messages are automatically archived and can be retrieved.

For messaging to work correctly in a CCTK project the STDMSG channel must be present and the *Message* process must be running. The *Message* process reads messages from STDMSG channel, interprets them, and forwards them to interested parties, including the system message file and the archive subsystem. Listing 3-37 provides a partial project configuration file showing the message configuration.

Listing 3-37: Sample messaging configuration

```
[… project configuration file details removed …]

<Channel name=”STDMSG” size=”10000” />

<Execute waitFor="running" critical="YES">Message</Execute>

[… project configuration file details removed …]
```

The STDMSG channel and the *Message* process must be configured for each CCTK project.

3.5.2 Archive

Archive stores measurement, command, and notice information for future retrieval. CCTK will automatically forward all changed measurements and all system notices to the STDARCH channel. For proper CCTK operation, a process must read the STDARCH channel to keep it from filling up. Typically, the *TamArs* process reads the STDARCH channel, processes archive packets, and stores them to the archive media. If, for a particular configuration, no data archive is necessary, CCTK should be configured to use a ChannelReader process to read STDARCH instead.

In most cases, an archive subsystem is desirable. To configure CCTK to use the archive subsystem, the following setup must exist:

- STDARCH channel must be defined in the project configuration file.
- *TamArs* must be executed in the project configuration file (see *TamArs* on page 162).
- *ArchiveControl* must be executed in the project configuration file (see *ArchiveControl* on page 116).
- A valid archive configuration file must exist in the project directory (see Archive Configuration on page 22).
- An ARS interface must be defined in the database (see *ARSInterface* in Section 3.4.10.1).

Table 3-36 provides a simple example of a partial project configuration file using *TamArs* and *ArchiveControl* for the archive subsystem.

Listing 3-38: Sample archive configuration using TamArs and ArchiveControl

```
[… project configuration file details removed …]

<Channel name=”STDARCH” size=”10000” />

<Execute waitFor="running" critical="YES">TamArs –f ars_config.d</Execute>
<Execute waitFor=”running” critical=”YES”>ArchiveControl</Execute>
```

```
[… project configuration file details removed …]
```

To configure CCTK without support for archive, the following setup must exist:

- STDARCH channel must be defined in the project configuration file.
- *ChannelReader* must be executed in the project configuration file.

Listing 3-39 provides an example of a partial project configuration file using *ChannelReader* for archiving.

Listing 3-39: Sample archive configuration using ChannelReader

```
[… project configuration file details removed …]

<Channel name=”STDARCH” size=”10000” />

<Execute waitFor="running" critical="YES">ChannelReader -c STDARCH -noout</Execute>

[… project configuration file details removed …]
```

See *ChannelReader* on page 126.

It is important to note that only one of the above methods can be configured at any one time. Both *ChannelReader* and *TamArs* cannot read the STDARCH channel at the same time. If the archive subsystem is needed, use the *TamArs* method; if no archive subsystem is needed, use the *ChannelReader* method. Please note that it does not matter how the STDARCH channel is read, as long as it is read. If the standard CCTK archive subsystem is not sufficient, it is possible to create a custom archive subsystem (possibly based upon an SQL database), by reading the data from the STDARCH channel and performing the necessary operations. The standard CCTK API provides the necessary support for building a replacement archive subsystem.

3.5.3 Commanding

CCTK provides underlying support for commands. For commands to operate correctly, several system resources must be configured. These resources include:

- STDRESP channel for command responses must be defined in the project configuration file.
- A single *DataProc*, reading STDRESP must be executed.

Listing 3-40 presents an example of a partial project configuration file with commanding configured.

Listing 3-40: Sample commanding configuration

```
[… project configuration file details removed …]
```

```
<Channel name=”STDRESP” size=”10000” />

<Execute waitFor="running" critical="YES">DataProc –i STDRESP –t mdt</Execute>

[… project configuration file details removed …]
```

The commanding configuration must be present for CCTK to operate properly.

3.5.4 Health and Status

Health and status is another basic service of CCTK. For health and status to operate correctly, several system resources must be configured. These resources include:

- STDSTATUS channel must be defined in the project configuration file.
- A single *DataProc*, reading STDSTATUS must be executed.

Listing 3-41 shows a sample health and status configuration.

Listing 3-41: Sample health and status configuration

```
[… project configuration file details removed …]

<Channel name=”STDSTATUS” size=”10000” />

<Execute waitFor="running" critical="YES">DataProc –i STDSTATUS –t mdt</Execute>

[… project configuration file details removed …]
```

The health and status configuration must be present for CCTK to operate properly.

3.5.5 Data Processing

Data processing processes measurement and command data within CCTK. Two processes within CCTK perform standard data processing:

- *PktDcom*: Breaks blocks of contiguous data into smaller components and tags the smaller components as measurements. *PktDcom* uses link descriptor records that are defined in the configuration database to control its operations. *PktDcom* can be used in many different situations; for example, *PktDcom* is typically used to decommutate PCM frames. See *PktDcom* on page 144.
- *DataProc*: Takes raw values of measurements and converts them into processed values while performing a variety of checks and comparisons. Each measurement in CCTK defines a set of data processing parameters in the configuration database. *DataProc* is the process that performs the operations associated with these parameters. See *DataProc* on page 131.

Data flows in to and out of both *PktDcom* and *DataProc* via channels. *PktDcom* takes two channel command line arguments, an input channel and an output channel. *DataProc* takes a single channel command line argument, an input channel. See the manual pages of both of these processes for details on the exact format of the command line arguments.

Each CCTK system runs zero or more *PktDcom* processes and one or more *DataProc* processes. External interfaces that generate blocks of data (such as PCM and IBS) typically pass data to *PktDcom*. Other external interfaces will pass data to *DataProc*. Please see the documentation on the individual external interfaces to see if they require *PktDcom* or *DataProc*. Each *PktDcom* must feed data to a *DataProc*. Figure 3-1 provides a simple graphical representation this concept.

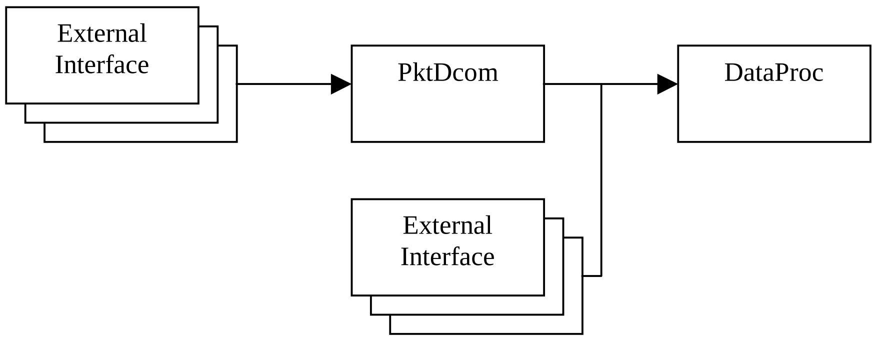


Figure 3-1: Data processing data flow via channels

When configuring a CCTK system, it is important to decide how many *PktDcom* and/or *DataProc* processes to execute. There are several advantages to running a single *PktDcom* and *DataProc*:

- Simpler configuration.
- Less overhead operating system overhead since there will be fewer task switches.

The following advantages are associated with multiple *PktDcoms* and multiple *DataProcs*:

- Processing can be performed in parallel across multiple processors to increase throughput.
- Problems in one processing thread will not affect other processing threads.

There is no simple way to determine the best configuration for every set of requirements. Please contact CCT for further assistance in designing a data processing scheme to meet your particular system needs.

Once the decision has been made on the number of *PktDcom* and *DataProc* processes to execute, the resources can be added to the project configuration file. Figure 3-2 shows a basic system with several external interfaces, a single *PktDcom*, and multiple *DataProcs*. Listing 3-42 provides an example of a portion of the necessary project configuration file to generate the above system configuration.

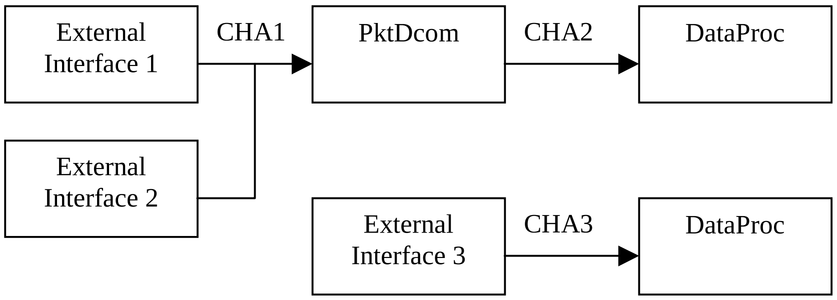


Figure 3-2: Example data processing configuration diagram

Listing 3-42: Sample data processing system configuration file

```
[… project configuration file details removed …]

<Channel name=”CHA1” size=”10000” />
<Channel name=”CHA2” size=”10000” />
<Channel name=”CHA3” size=”10000” />
<Execute waitFor="running" critical="YES">DataProc –i CHA2 –t mdt</Execute>
<Execute waitFor="running" critical="YES">DataProc –i CHA3 –t mdt</Execute>
<Execute waitFor="running" critical="YES">PktDcom –i CHA1 –o CHA2</Execute>
<Execute waitFor="running" critical="YES">EndItemInteface1 –o CHA1</Execute>
<Execute waitFor="running" critical="YES">EndItemInteface2 –o CHA1</Execute>
<Execute waitFor="running" critical="YES">EndItemInteface3 –o CHA3</Execute>

[… project configuration file details removed …]
```

Please note that the order of the data processing elements specified in the project configuration file is important as the channel client (reader) must be executed before the server (writer). Thus, *DataProc* must be executed before *PktDcom*, which must be executed before any external interfaces.

DataProc will forward processed data to other applications within the CCTK system over channels using one of two methods. First, it is permissible for any application to register to receive data via a channel. When registering for the data, a channel is specified where the information is received. No project configuration is required to support this type of data forwarding. Second, it is permissible for any application to receive blocks of processed data from *DataProc*. *DataProc* will write out packets of data identical to the type that it receives to the channel specified on the command line with the –o option. Thus if *DataProc* receives superpackets of linked data packets on its input channel, it will send superpackets of processed data on its output channel. This option can be used to support time correlated data.

For example, if an interface is capable of identifying time correlated data, it could create a superpacket of linked data packets containing the time correlated data. The superpacket could then be sent to *DataProc*. *DataProc* would process the data normally, updating the real-time tables, and distributing the data via channels. In addition, if the –o option was specified, *DataProc* would send a superpacket of processed data to the channel specified by the –o option. Thus, an application requiring time correlated data could read the data from that channel and process the data as a group.

Furthermore, *PktDcom* will send a superpacket of linked data packets to *DataProc* for each block it receives. Thus it is possible to receive blocks of time correlated data if *PktDcom* is used to process the data from an interface whose data is time correlated blocks of data.

Figure 3-3 graphically shows these processing scenarios while Listing 3-2 shows the associated configuration.

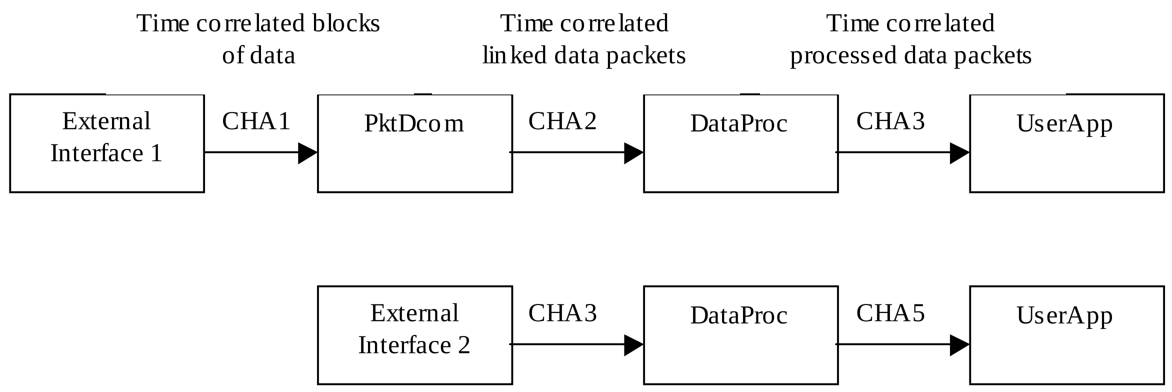

Time correlated blocks of data
Time correlated linked data packets
Time correlated processed data packets
External Interface 1
CHA1
PktDcom
CHA2
DataProc
CHA3
UserApp
External Interface 2
CHA3
DataProc
CHA5
UserApp

Figure 3-3: Example data processing configuration diagram for time correlated data

Listing 3-43: Sample data processing system configuration file for time correlated data

```
[… project configuration file details removed …]

<Channel name=”CHA1” size=”10000” />
<Channel name=”CHA2” size=”10000” />
<Channel name=”CHA3” size=”10000” />
<Channel name=”CHA4” size=”10000” />
<Channel name=”CHA5” size=”10000” />

<Execute waitFor=”running” critical=”YES”>UserApp –I CHA3</Execute>
<Execute waitFor="running" critical="YES">DataProc –i CHA2 –o CH3 –t mdt</Execute>
<Execute waitFor="running" critical="YES">PktDcom –i CHA1 –o CHA2</Execute>
<Execute waitFor="running" critical="YES">EndItemInteface1 –o CHA1</Execute>

<Execute waitFor=”running” critical=”YES”>UserApp –I CHA5</Execute>
<Execute waitFor="running" critical="YES">DataProc –i CHA4 –o CHA5 –t mdt</Execute>
<Execute waitFor="running" critical="YES">EndItemInteface2 –o CHA4</Execute>

[… project configuration file details removed …]
```

3.5.6 Time Control

CCTK provides underlying support for countdown and project elapsed time control. For countdown time control to work properly several resources must be configured. These resources include:

- A channel must be configured to transmit time control data.

- A *DataProc* must be configured to process time control data.
- *CDTimeControl* process must be executed (See *CDTimeControl* on page 119).
- A time control interface must be defined in the database (see Section 3.4.10.2 for information on defining a time control interface in the configuration database).

Listing 3-44 provides an example of configuring time control.

Listing 3-44: Sample time control configuration

```
[… project configuration file details removed …]

<Channel name=”TimeControlData” size=”10000” />

<Execute waitFor="running" critical="YES">DataProc –i TimeControlChan –t mdt</Execute>
<Execute waitFor="running" critical="YES">
             CDTimeControl -p CDTime_port -o TimeControlChan -u 1000</Execute>

[… project configuration file details removed …]
```

Time control is not a required part of CCTK. If the above configuration is omitted, time control services will not be available.

3.5.7 Multicasting Data

CCTK provides a generic mechanism for multicasting data to clients on the network. See *MulticastServer* on page 139. *MulticastServer* is a legacy application and will be replaced in the future by *PeerSender*. If a choice is available, please use *PeerSender* instead of *MulticastServer*.

If multiple CCTK projects are going to simultaneously execute on the same network and those projects are going to each run an instance of *MulticastServer*, special care must be taken to prevent multicast address conflicts. Each *MulticastServer* must use a different address/port to transfer data to and from the clients. Multiple *MulticastServers* cannot be started on the same system unless different address/port combinations are used. Such an attempt will cause the second server to fail. If multiple *MulticastServers* with the same address are started on a single network on different hosts, no errors will result, but unpredictable results will occur.

The command line arguments –m (for address) and –p (for port) can be used to alter the multicast address/port combination. If the –m and –p options are not given to *MulticastServer*, then the defaults will be used. The default address is 239.1.1.1 and the default port is 7777.

It is also possible to start multiple *MulticastServers* from within a single project. Listing 3-45 shows an example *MulticastServer* configuration where one server is configured to use the default address/port while another is configured to use a different address/port pair.

Listing 3-45: Sample multicast server configuration

```
[… project configuration file details removed …]
```

```
<Channel name=”TimeControlData” size=”10000” />

<Execute waitFor="running" critical="YES">MulticastServer</Execute>
<Execute waitFor="running" critical="YES">MulticastServer –p 7000 –m 239.1.1.2</Execute>

[… project configuration file details removed …]
```

Multicast is not a required part of CCTK. The *MulticastServer* process must be running to feed data to the CCTK *StripChart* client. If the above configuration is omitted, multicast and thus the *StripChart* client will not work, but the remainder of CCTK will operate properly.

3.5.8 Peer-to-Peer Interface

The CCTK peer-to-peer interface allows for CCTK to send/receive data over a network. *PeerSender* transmits data from CCTK to a receiving peer; *PeerReceiver* receives data into CCTK from a sending peer. It is possible to configure two or more CCTK projects to transfer data by connecting a CCTK *PeerSender* on one project to a CCTK *PeerReceiver* on another project. Since the peer to peer protocol is open, it is also possible to write custom senders and receivers to interface other systems with CCTK. The peer to peer protocol is documented in the PeerProtocol.h header file found in the CCTK include directory. For more information on the protocol, see this file or contact CCT for more information.

Both *PeerSender* and *PeerReceiver* support multiple modes of operation based upon the type of socket desired for data exchange. Table 3-48 shows the four primary connection types supported by *PeerSender* and *PeerReceiver*.

Table 3-48: PeerSender/PeerReceiver Modes

Mode	Socket Type	Protocol	Notes	Example
Point-to-Point / Registration	Stream	TCP	Provides a point to point connection between two systems. Transmission is guaranteed. Packets can easily be routed. High latency is possible if many packets are lost. Two way communication is permitted with point-to-point.	Two systems need to exchange data across a WAN. Data integrity must be guaranteed, but latency is not an issue.
Connectionless	Datagram	UDP	Provides a communication path between two systems. Transmission is not guaranteed. Packets can easily be routed. Latency still an issue but not as significant a factor as in a point-to-point connection especially if a dedicated LAN is used.	Two systems need to exchange data across a WAN. Data latency is important, some data can be lost. Two systems need to exchange data on a dedicated LAN.
Broadcast	Datagram	UDP	Provides a communication path between one sender and multiple receivers. Packets cannot be routed. Latency is typically not an issue as data is transmitted on a dedicated LAN.	A single system will feed identical data to multiple systems. (i.e. a display server feeding data to multiple displays).

Mode	Socket Type	Protocol	Notes	Example
Multicast	Datagram	Multicast	Provides a communication path between one sender and multiple receivers. Packets can be routed.	A single system will feed identical data to multiple systems. Systems are on the Internet. This method is typically used when large amounts of data need to be moved to multiple disjoint systems (streaming audio/video).

3.5.8.1 PeerSender

PeerSender reads a CCTK channel and transmits the data it receives via a socket. *PeerSender* configures the socket based upon information either on the command line or in the *PeerSenderInterface* descriptor. If the configuration information is specified in a descriptor, the descriptor name must be provided to *PeerSender* on the command using the –P flag. If a descriptor is specified, it is not permissible to configure the socket via the command line. If no descriptor is specified, then the socket must be configured via the command line. The required socket arguments vary depending upon the mode of operation. The *PeerSender* manual page, found in Section 8, provides details on the command line options.

If *PeerSender* is configured with a descriptor, commanding is activated automatically. *PeerSender* can be commanded to start and stop data transmission using the appropriate system commands. By default these commands are called "<name> - Start DT" and "<name> - Stop DT", where <name> is the name of the *PeerSenderInterface* descriptor. These names can be overridden in the *PeerSenderInterface* descriptor. *PeerSender* defaults to data transmission stopped at startup.

PeerSender receives data to send via a CCTK channel. The channel is a named channel when operating in point-to-point, connectionless, broadcast, or multicast modes. In these modes, the channel must be specified on the command line using the –i option and one or more CCTK processes should be configured to feed data to *PeerSender*. Registration mode is a unique mode for *PeerSender*. When using registration mode, a named input channel may be specified, but it is optional. The input channel specified on the command line will only be used for sending and receiving commands. When operating in registration mode, *PeerSender* will open the next available channel for data reception each time a new peer connects. *PeerSender* expects the receiving peer to send registration requests on which measurements/notices it wishes to receive. When a registration request is received, *PeerSender* will register for that measurement with the CCTK system. At that point, all changed values for that measurement will be sent to next available channel and then forwarded on the socket to the receiving peer. This mode provides a way in which multiple receivers can connect to the CCTK system and request which measurements/notices they want.

In addition to data packets, PeerSender will also send identification packets across the socket. An identification packet is used by the receiver to correlate the information in the data packets with the name of the descriptor on the sending system. **Each identification packet is only sent the first time a sender sees the packet. Therefore, it most situations, it is important that the receiver be active prior to the sender**.

3.5.8.2 PeerReceiver

PeerReceiver reads a socket and transmits the data it receives via a CCTK channel. *PeerReceiver* configures the socket based upon information either on the command line or in the *PeerReceiverInterface* descriptor. If the configuration information is specified in a descriptor, the descriptor name must be provided to *PeerReceiver* on the command using the –P flag. If a descriptor is specified, it is not permissible to configure the socket via the command line. If no descriptor is specified, then the socket must be configured via the command line. The required socket arguments vary depending upon the mode of operation. The *PeerReceiver* manual page, found in Section 8 provides details on the command line options.

If *PeerReceiver* is configured with a descriptor, commanding is activated automatically. *PeerReceiver* can be commanded to start and stop data acquisition and data processing using the appropriate system commands. By default these commands are called "<name> - Start DA" and "<name> - Stop DA" for data acquisition and called "<name> - Start DP" and "<name> - Stop DP" for data processing, where <name> is the name of the *PeerRecieverInterface* descriptor. These names can be overridden in the *PeerReceiverInterface* descriptor. *PeerReceiver* defaults to data acquisition and data transmission stopped at startup.

PeerReceiver sends data to the CCTK over a named output channel. The output channel must be present on the command line and is specified by the –o argument. A client must be configured to read the channel. The client must be able to understand the packets being sent to it via the peer interface. For example, if a *PeerSender* is reading the output channel of *PktDcom*, then the *PeerReceiver* receiving the data should forward it into a *DataProc*. *PeerReceiver* also has a special translation mode where incoming processed data packets can be converted to linked data packets. This mode can be used to allow processed data packets received from a *DataProc* to be retransmitted through the CCTK system. This translation is activated using the –p command line argument.

For *PeerReceiver* to operate correctly, it must be able to translate the names of the incoming descriptors to names valid on its local system. It can do this in one of two ways. First, if the names of the descriptors are identical on the two systems, then a direct name translation can be performed. Second, if the names of the descriptors are different on the two systems, then a map can be used to map the names on the sender system to valid names on the receiver system. The map can be configured as described in section 3.4.10.4. For *PeerReceiver* to have a map, it must be configured via a *PeerReceiverInterface* descriptor. By default, *PeerReceiver* will attempt to translate the measurement via the map and then via direct name translation.

3.5.8.3 Sample Peer to Peer Configurations

As noted earlier, registration (or server) mode causes *PeerServer* to operate as a traditional UNIX server similar to ftp, http, rsh, etc … For *PeerServer* to operate in registration mode, the local port must be specified. This is the port that the receiving peers will use to establish a connection (i.e. http uses port 80). With each new connection, a new instances of *PeerServer* will be created to handle the connection. The receiver can register for measurements/notices that *PeerServer* will transmit over the socket. Listing 3-46 shows a basic CCTK project configuration file with *PeerServer* operating in registration mode.

Listing 3-46: Sample *PeerServer* registration mode configuration

```
[… project configuration file details removed …]

<Execute waitFor="running" critical="YES">PeerSender –m registration –L 10000</Execute>

[… project configuration file details removed …]
```

In this example, two CCTK systems communicate using a connectionless socket. This is typically used when two CCTK systems that reside on a dedicated network need to share data. One possible use for this type of configuration is to distribute the processing load across multiple hosts. In this example, the output of *PktDcom* on the sending host is forwarded across the link to be processed by a *DataProc* on the receiving host. It is assumed that the database has been properly configured so that the measurements map correctly.

Listing 3-47 shows the project configuration file on the sender. Listing 3-48 shows the project configuration file on the receiver. In this example, the IP address of the sender is 192.168.1.1 and the IP address of the receiver is 192.168.1.1. *EndItemIf* is a hypothetical end-item interface process that feeds packets to *PktDcom*.

Listing 3-47: Sample *PeerServer* connectionless mode configuration

```
[… project configuration file details removed …]

<Execute waitFor="running" critical="YES">PeerSender -i PktDcomOut –m connectionless –r 
192.168.1.2 –R 10000</Execute>
<Execute waitFor="running" critical="YES">PktDcom -i PktDcomIn -o PktDcomOut</Execute>
<Execute waitFor=”running” critical=”YES”>EndItemIf –o PktDcomIn</Execute>

[… project configuration file details removed …]
```

Listing 3-48: Sample *PeerReceiver* connectionless mode configuration

```
[… project configuration file details removed …]

<Execute waitFor=”running” critical=”YES”>DataProc –I DataProcIn –t mdt</Execute>
<Execute waitFor="running" critical="YES">PeerReceiver –m connectionless –o DataProcIn –L 
10000</Execute>

[… project configuration file details removed …]
```

In this final example, a *PeerReceiver* is configured to receive processed data packets via a broadcast address. Since *PeerReceiver* is feeding the data into a *DataProc*, the packets must be converted to linked data packets. The –p option enables this translation. Listing 3-49 shows a sample project configuration file.

Listing 3-49: Sample *PeerReceiver* broadcat mode configuration

```
[… project configuration file details removed …]

<Execute waitFor=”running” critical=”YES”>DataProc –i DataProcIn –t mdt</Execute>
<Execute waitFor="running" critical="YES">PeerReceiver -p –m broadcast –o DataProcIn –L
10000</Execute>

[… project configuration file details removed …]
```

3.5.9 External Interfaces

In most cases, it is necessary to run one or more external interfaces to make CCTK useful. External interfaces typically read data from external hardware and forward it to CCTK. External interfaces also receive commands from CCTK and alter the state of the hardware to reflect the command. Several different interface types exist for CCTK. For information on configuring these interfaces, please see the documentation associated with the proper interface. In addition, it is possible to add new interfaces to CCTK programmatically. Please see the *CCTK Developer’s Guide* for more information on adding custom interfaces.

3.6 Creating a CCTK Project

This section unifies the previous discussions on project directory, project configuration, configuration database, and configuration of system tasks into a single discussion on how to create a working CCTK project. This section presents, in simple terms, those resources that are required and those resources that are optional when creating a CCTK project.

As discussed previously, each CCTK project is contained within a single directory, the project directory. The project directory can reside anywhere on the local file system of the CCTK server. It is created using standard UNIX commands. Permissions on the project directory should be set appropriately. See Section 3.1 for more information on the project directory.

Within the project directory, two files must be created:

- Project configuration file (see Section 3.2 for more details).
- Configuration database file (see Section 3.4 for more details).

Optionally, the following files may be created:

- Additional configuration database files.
- Archive configuration file (see Section 3.3 for more details).
- Other files as required by specific CCTK processes.

Listing 3-50 shows a minimal project configuration file.

Listing 3-50: Simple project configuration file

```
<?xml version="1.0"?>
<!DOCTYPE ProjectConfiguration SYSTEM "project_config.dtd">
```

```
<ProjectConfiguration project_name="Simple Project">
    <StartUp defaultMode="simple">
        <Mode name="simple">
            <ConfigDb file="simple_config_db.xml" />
            <StatusTable entries="20"/>
            <Channel name="STDMSG" size="40000" />
            <Channel name="STDRESP" size="40000" />
            <Channel name="STDDIST" size="40000" />
            <Channel name="STDSTATUS" size="40000" />
            <Channel name="STDARCH" size="40000" />
            <Execute waitFor="running" critical="YES">Message</Execute>
            <Execute waitFor="running" critical="YES">DataProc -i STDRESP -t
mdt</Execute>
            <Execute waitFor="running" critical="YES">DataProc -i STDRESP -t
mdt</Execute>
            <Execute waitFor="running" critical="YES">TamArs</Execute>
            <Execute waitFor="running" critical="YES">ArchiveControl</Execute>
        </Mode>
    </StartUp>
</ProjectConfiguration>
```

Within this project configuration file, the following required resources must be defined:

- Project name.
- A single mode.
- Status table entries.
- Minimum set of CCTK channels (STDARCH, STDRESP, STDSTATUS, and STDMSG).
- A single configuration database file.
- Only those processes that are necessary to run a simple CCTK system, including *Message*, a *DataProc* reading STDRESP, a *DataProc* reading STDSTATUS, and archive processes.

Listing 3-51 shows a minimum configuration database file.

Listing 3-51: Simple configuration database

```
<?xml version="1.0"?>

<!DOCTYPE ConfigurationDatabase SYSTEM "xcdb_config.dtd" [
    <!ENTITY SystemNotices SYSTEM "SysNotice.xml">
    ]>

<ConfigurationDatabase>
    <Table name="ndt" />
    <Table name="mdt" />
    <Table name="cdt" />
    <Table name="ldt" />
    <Table name="pdt" />
    <Table name="bdt" />
    &SystemNotices;
    <ArsInterface name="ARS_port_descriptor">
        <ArsStartCommand name="Start_ARS" />
        <ArsStopCommand name="Stop_ARS" />
        <ArsCloseArchiveCommand name="ARS_Close_Archive" />
        <ArsStatusMeasurement name="ARS_Status" table="mdt"/>
    </ArsInterface>
</ConfigurationDatabase>
```

Within this configuration database file, the following required resources must be defined:

- Core CCTK tables.
- Included system notices.
- ARS Interface.

The project configured above is not very useful in and of itself, but it does provide a starting point for future projects and shows the minimal amount of configuration necessary for a running CCTK system.

4 PROJECT EXECUTION

This section discusses a variety of topics related to project execution. Project execution includes all activities that occur between and including CCTK startup and shutdown. The following essential items will be examined:

- Startup and Shutdown: This section discusses *ProjectManager*, the back-end process called by *CctkClient* starts up and shuts down a project. Basic project startup and shutdown using *CctkClient* is discussed in the *CCTK User's Manual*.
- Starting and Stopping Applications: It is possible to start and stop certain applications while the project is executing. This section discusses the pros and cons of performing these actions.
- Project Directory: This section discusses the files and directories created in the project directory during project execution.
- Health and Status: This section shows how health and status is calculated as well as how to obtain the current health and status of the system and individual applications within the system.
- Utility Applications: CCTK has several utility applications that can be run during project execution. These utilities are very useful when debugging problems. This section introduces these utilities.

4.1 Starting and Stopping a Project

ProjectManager is a command line application used to start and stop a project. *ProjectManager* has many command line options that control its behavior. The following subsections discuss the different options passed to *ProjectManager* and how they can be used to control project startup and shutdown.

CctkClient provides a simple to use (although less powerful) graphical front-end to *ProjectManager*. The *CCTK User's Manual* describes how to use the *CctkClient* program to start and stop a project.

4.1.1 Using ProjectManager to Start a Project

Executing *ProjectManager* from the command line will start a CCTK project. If no arguments are passed to *ProjectManager*, then it will attempt to derive all of the necessary information needed to start CCTK.

```
% ProjectManager
Initializing the CCTK system...
      [… output dependent upon project configuration …]
Creating Status Table...
      [… status table output dependent upon project configuration …]
Creating Environment Variables...
      [… environment variable output dependent upon project configuration …]
Creating Channels...
      [… channel output dependent upon project configuration …]
Creating Message Queues...
      [… message queue output dependent upon project configuration …]
Building Real-time Tables...
      [… real-time table output dependent upon project configuration …]
Executing Processes...
      [… process execution output dependent upon project configuration …]
%
```

A particular project mode can be specified using the '-m' option:

```
% ProjectManager –m requested_mode
      [… output similar to previous example …]
%
```

In both of the above situations, *KPATH* must not be set and the project configuration file must be named the default name of *project_config.pcml*. In this situation, *ProjectManager* will deduce that the current directory is to be the project directory and that the configuration file to use is *project_config.pcml*.

ProjectManager uses the following rules to determine the project directory and configuration file:

- If -k is specified on the command line, it is used.
- If -k is not specified, but the *KPATH* environment variable exists and is valid, it is used.
- If neither -k nor the *KPATH* environment variables are specified, *ProjectManager* will attempt to deduce the project directory.
  - If a configuration file (other than the default) is specified on the command line with the -f option, *ProjectManager* will assume that the project directory is the directory where the configuration file resides.
  - If no configuration file is specified on the command line, then *ProjectManager* will assume that the current directory is the project directory and attempt to use it.

For example, if the following command is executed:

```
% ProjectManager -k /some/valid/directory
       [… output similar to previous example …]
%
```

Then *ProjectManager* will use the given directory as the project directory. If the *KPATH* environment variable is set, as in the following example:

```
% echo $KPATH
/some/valid/directory
% ProjectManager
       [… output similar to previous example …]
%
```

Then *ProjectManager* will use the directory defined by *KPATH* as the project directory. As discussed above, if no arguments are given:

```
% ProjectManager
       [… output similar to previous example …]
%
```

Then *ProjectManager* will assume that the current directory is the project directory and act appropriately.

Once *ProjectManager* deduces a project directory, it checks to ensure that the directory exists and that the user has execute and write permissions to the directory. If the user does not have the appropriate permissions, *ProjectManager* will fail. *ProjectManager* will also verify that the configuration file project_config.pcml (unless specified differently with the -f command line option) exists and is readable by the user. If *ProjectManager* cannot find an appropriate configuration file or if the user cannot read it, *ProjectManager* will fail.

ProjectManager will only attempt to perform a startup if the project is in a down state. Valid down states include “DOWN”, “FORCED_DOWN”, and “STARTUP_FAILED”. If the project is not in a down state, *ProjectManager* will fail to successfully start the system. If the project is not in a down state and will not shutdown correctly, Section 4.1.3 provides information on cleaning up project resources and resetting the system state.

Table 4-1 lists *ProjectManager* command line options for project startup.

Table 4-1: ProjectManager startup options

Option	Description
-u	Bring the project up (this is the default and thus not necessary).
-f config_file	Use config_file as the project configuration file.
-k kpath	Use kpath as the project configuration directory.
-m mode	Bring up the project in the specified mode. The mode should exist in the project configuration file.
-n user_notes	Enter the user_notes into the state history file.

Option	Description
-q	Only display error messages when starting the project, informational messages will be surpressed. This option is exclusive with the –v option.
-v	Produce verbose output when starting the project. This option is exclusive with the –q option. This is the default.
-c	Clean up the log directory and tmp directory before starting the project.
-h	List the full set of options on the command line with a short help message on each option.

See *ProjectManager* on page 163.

4.1.2 Using ProjectManager to Stop a Project

A project is stopped or shutdown by sending the UNIX TERM signal to the *ProjectManager* which started the project. If the user has only one project running on the system or wishes to terminate all projects he/she is running on the a system, the following UNIX command can be used:

```
% killall –TERM ProjectManager
%
```

If only one project needs to be shutdown, then the PID of the associated *ProjectManager* must be determined. This is determined by examining the PROC directory in the project's temporary directory. Within the PROC directory, an entry for *ProjectManager* should exist and the id should be part of the name. The TERM signal can then be sent to the appropriate process using the UNIX *kill* command. This process is automated using *ProjectManager*.

```
% ProjectManager –d -D
Shutting down the CCTK system...
       [ … shutdown information specific to the particular project …]
%
```

The above command will cause *ProjectManager* to look for a running *ProjectManager* and attempt to terminate it. For the above command to work, *ProjectManager* must be able to determine the project directory. The same rules described in Section 4.1.2 apply. In addition, the –D option instructs *ProjectManager* to terminate all applications associated with this project, not just the ones it started. Other options are listed in Table 4-2.

As with system startup, *ProjectManager* will check the state before performing a shutdown. If the state is not "UP", the shutdown will fail. At times, it may be necessary to force a project down regardless of state. Adding the –i option to *ProjectManager* will force a project down.

Table 4-2: *ProjectManager* shutdown options

Option	Description
-d	Bring the project down
-f config_file	Use config_file as the project configuration file.
-k kpath	Use kpath as the project configuration directory.
-n user_notes	Enter the user_notes into the state history file.
-D	Send the terminate signal to all processes associated with the project, not just those started by ProjectManager.
-i	Force a shutdown regardless of state.
-q	Only display error messages when starting the project, informational messages will be surpressed. This option is exclusive with the –v option.
-v	Produce verbose output when starting the project. This option is exclusive with the –q option. This is the default.
-c	Clean up the log directory and tmp directory before starting the project.
-h	List the full set of options on the command line with a short help message on each option.

4.1.3 Cleaning Up Project Resources

If problems occur during project execution or termination, it is possible that the CCTK project is in an unusable state. If, for example, the temporary directory is deleted while the project is running, *ProjectManager* may not be able to cleanup the resources associated with a project.

When a project does not shutdown properly, processes, shared resources, or both remain active on the system. These processes and/or shared resources must be cleaned up before the project can successfully execute again. The UNIX *ps* command and *ipcs* command can be used to view processes and shared resources on the system. If you are unfamiliar with these commands, reference the UNIX documentation that came with your system.

The following steps should clean up all resources associated with the project and allow it to execute again. Be sure that you have changed to the project directory and that *KPATH* is set before executing these steps.

First, attempt to stop the project using the normal shutdown command.

```
% ProjectManager -dD
```

If that does not work, attempt to force the project down using the force option of shutdown.

```
% ProjectManager -dDi
```

If a project is still up, use *ps* to list all of the processes belonging to the user that started the project. Look for the core CCTK processes (such as *DataProc*, *ProjectManager*, *Message*, etc.) and note their PID’s. Using the UNIX *kill -9* command to destroy these processes. If the same user has multiple projects active, be certain to terminate only the processes associated with the problem project. After cleaning up the processes, use the *ipcs* command to view the shared resources belonging to this user. Use the *ipcrm* command to remove shared resources associated with this user. Again, if the same user has multiple projects active, be careful to terminate only the resources associated with the problem project.

Once the processes and shared resources are cleaned up, remove the temporary directory using the appropriate UNIX command.

Rerun the last *ProjectManager* command described above to ensure that the system is placed in a down state.

If, after following the above steps, the CCTK project will still not start up, please contact CCT for support.

4.2 Starting and Stopping Applications

Several programs are provided as part of CCTK. These programs, sometimes referred to as "system applications" or "graphical user interfaces," are general-purpose tools available to all users for performing common tasks. The names of the executable programs associated with these CCTK applications are listed in Table 4-3 below. You can also create custom applications using the optional CCTK development environment.

Table 4-3: System applications available during project execution

Name	Executable	Description
CCTK Client	*CctkClient*	Top-level interface to CCTK. Allows the user to select a project, start and stop a project, view displays, and launch applications.
Measurement Monitor	*MeasMon*	Allows the user to view the current value of measurements in the system. Also allows the user to view properties of the measurement such as status, units, and last processed time.
Retriever	*Retriever*	Allows the user to retrieve historic information from a project's archive. Measurements, commands, and messages can be retrieved.
System Message	*SysMsgGui*	Allows the user to view the system messages associated with a project.
Time Control	*N/A*	Allows the user to monitor and control CDT, including scheduling CDT start and stop events based on time.
Simulator	*SimGUI*	Allows the user to create and edit CCTL simulation scripts, and control the simulation during project execution.

After a CCTK project has been started, it is common to start one or more applications (including their graphical displays) which communicate with the CCTK project. To start a CCTK application (user or system), the *KPATH* environment variable must be set to the project directory and the user must have permission to communicate with the project. Applications and displays are typically started from the command line or via an automated remote login from a client (see Section 4.3 for more information on this method). It is also common to start applications from the *CctkClient* display tree. In this case, *CctkClient* sets *KPATH* based upon the opened project. See Section 3.2.3.2 for information on adding applications to the *CctkClient* display tree.

4.3 Connecting to the Server From a Client

When operating CCTK in the client/server environment, it is necessary to execute applications remotely on the server and display the applications on the local client display.

The features of X Window allow applications running on the CCTK server to be displayed on the local client display. In order for X Window to operate properly, the *DISPLAY* environment variable must be set to the correct host before starting the X Window application. Reference the X Window documentation for more information.

Remote execution can be accomplished using several different methods, but the most common method is to use the UNIX remote execution utilities (*rsh* and/or *rexec*). Please see the UNIX documentation for configuring and using these utilities.

If Windows clients are being used with the Exceed X Window package, excellent documentation is available from Exceed on performing the above tasks.

4.4 Project Directory

Section 3.1 introduces the project directory. It discusses the basics of the project directory and describes the configuration files that reside in the project directory. In addition to these configuration files, a significant number of files and directories are created during project execution. This section discusses the files and directories generated in the project directory during project execution.

The following files/directories are created during project execution. Each of these is described in the following sections.

- Log Directory
- Temporary Directory
- Lists Directory
- PROC/KSHM/KMSG Directories
- System State File

4.4.1 Log Directory

The log directory ($*KPATH*/log) contains log files for all CCTK processes. The log directory is created upon project startup if it does not already exist. All CCTK processes create a log file in the log directory in which they write a variety of information including system messages, health and status, and occasionally debug output. These log files remain until deleted, either manually or through the use of *ProjectManager's* –c option.

The format of the log file names is:

<process name>.<UNIX PID>

Where process name is the internal name of the process, this is typically the same name as the executable, and UNIX PID is the UNIX process identifier.

If problems are encountered during CCTK operation, scanning the log files is one of the first troubleshooting steps. Most problems encountered during system execution are logged to the log files.

In addition to the process log files, a listing of the system notices processed by the system is stored in the log directory. *${KPATH}/log/system_messages.d* contains all of the system notices received by the system since it started. This text file is used by the system message GUI. It can also be viewed from the command line for a quick look at the system notices.

4.4.2 Temporary Directory

The temporary directory *(${KPATH}/tmp*) is created upon project execution if it does not already exist. The temporary directory contains temporary files directly related to the execution of the current project. Files within this directory typically have no relevance after the project stops. The temporary directory will be automatically cleared when the next project starts. Many different CCTK processes create files in the temporary directory. The temporary directory can be safely removed as long as the project is in the down state. The next few sections describe some of the files found in the project's temporary directory.

4.4.3 Lists Directory

One of the items in the temporary directory is the lists directory (*${KPATH}/tmp/lists*). The lists directory contains lists of function designators (FD's) for an active CCTK project. During database creation, lists are created for a variety of different descriptor types. All of these lists are present in this directory. The lists are used by a variety of CCTK processes for many different purposes. Manually viewing the lists is an excellent way to determine if an FD configured in the database was correctly created in the system. Table 4-4 shows all of the possible lists created by the CCTK database.

Table 4-4: CCTK database generated lists

FD Type	Filename
Analog Measurements	analog.list
Bus Descriptors	bus.list
Byte Array Measurements	bytearray.list
Commands	command.list
Command Responses	commandresponse.list
All Descriptors	descriptor.list
Discrete Measurements	discrete.list
Measurements	measurement.list
Notice Descriptors	notice.list
Port Descriptors	port.list
String Measurements	string.list
System Commands	systemcommand.list
Unsigned Integer Measurements	unsignedint.list

4.4.4 PROC/KSHM/KMSG Directories

The *PROC*, *KSHM*, and *KMSG* directories also reside in the tmp directory. These three directories all provide information on resources used by the CCTK project. They are only valid during project execution. The directories are not typically accessed during normal CCTK administration, but they must be present for the system to operate properly.

The *PROC* (*${KPATH}/tmp/PROC*) directory contains a single entry for each process attached to CCTK. Whenever a new process is started, it automatically adds an entry to the *PROC* directory. When the process is stopped, it automatically removes itself from the *PROC* directory. This directory can be used to associate running processes with a project. The entries in the PROC directory are formatted as <process name>.<pid>.

The *KSHM* (*${KPATH}/tmp/KSHM*) directory stores information on the shared memory segments created for the CCTK project. The name of the each file in the *KSHM* directory equates to the name of the shared memory segment. The file contains the shared memory key associated with the segment. Using this key, it is possible to associate the shared memory segments listed by the UNIX *ipcs* command with those used by a CCTK project.

The *KMSG* (*${KPATH}/tmp/KMSG*) directory stores information on the message queues created for the CCTK project. The description for the *KSHM* directory applies to the *KMSG* directory as well.

4.4.5 System State File

The system state file (*${KPATH}/system_state.xml*) resides in the project directory. It is an XML file that holds information on the state of CCTK. It is updated primarily by *ProjectManager* and read by a variety of different system applications. The system state file is a persistent file and will remain in the project directory even when the project is down. If the system state file is not present, it will be automatically created. This file can be safely removed as long as the project is down, but removing the file will erase all state history information.

The system state file contains the following information:

- Current state of the system
- Current state information
- Historical system state information

The contents of the system state file is used by *CctkClient* to provide much of the system information seen on its display. The state history widget obtains all of its historic information from this file. The state history widget also allows a user to prune and clear the state history. The *SystemState* utility application is a command line utility for viewing and modifying system state. The *SystemState* utility is described on page 161. The *SetSystemState* utility can be used to modify the system state file while the system is down. It is described on page 155. *ClearSystemStateOnBoot* is an additional utility provided to manage the system state. If you execute this script on boot up with the appropriate arguments, it will ensure that the specified projects are in the down state. This is useful for turn-key systems where you need to ensure that the projects are in a state that can be started when the system boots after a power outage or crash. *ClearSystemStateOnBoot* is described on page 130.

Since the system state file is an XML file, it is described by a DTD which is located in *${CCT_HOME}/include/dtd/system_state.dtd*.

4.5 Health and Status

CCTK has an integrated health and status system that continually monitors all processes associated with a CCTK project. The health of the CCTK project will be derived anytime the system is in the UP state. Two primary conditions cause a project's health to change:

- A process is unresponsive or exits improperly.
- A process alters its individual health status to cautionary or critical.

Each process in the CCTK maintains a health count. The process is responsible for incrementing the health count once a second. In addition, each process may optionally post its internal health. *ProjectManager* monitors the health counts and internal health of all processes in the system. If a health count stops incrementing, *ProjectManager* will change the health of the system to reflect the problem. If the health of an individual process alters from the healthy state, then *ProjectManager* will alter the health of the overall system to reflect this change.

The overall health of the system is displayed in the *CctkClient* status bar. A curses based utility called *StatMon* is available. *StatMon* shows the individual health, state, health counts, and other information for each process attached to the CCTK project. Section 4.6.2 provides instructions for executing *StatMon*.

4.6 Utility Applications

CCTK has several command line utility applications for interacting with the system. The following sections describe several of these utilities.

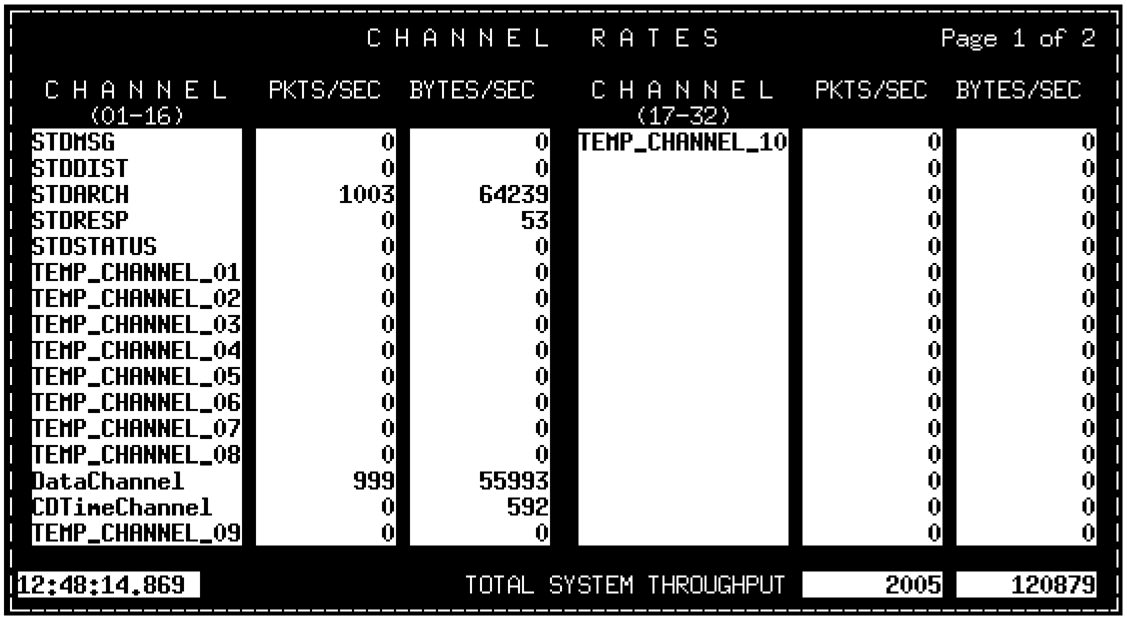
4.6.1 Channel Monitoring

CCTK uses a shared memory based IPC mechanism called channels to move blocks of data between processes. Correct operation of channels is an important aspect to a properly running CCTK system. *ChanMan* is a simple curses based utility that allows monitoring of channel statistics.

A project must be up and running and the *KPATH* must be set before executing *ChanMan*. *ChanMan* can be executed from the UNIX command prompt:

```
% ChanMan
```

To obtain a listing of the keystrokes used to navigate in *ChanMan*, press the 'h' key. To make the help screen go away, press the space bar. Figure 4-1 displays the channel rates *ChanMan* screen. This is the initial screen you will see after running *ChanMan*.



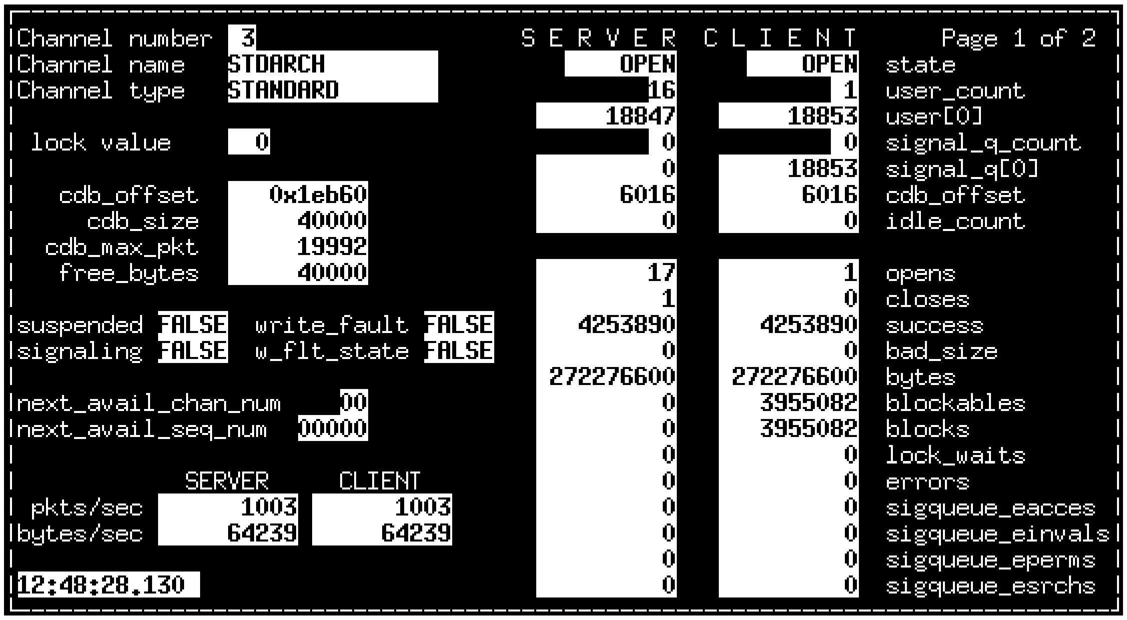
```
                        C H A N N E L   R A T E S              Page 1 of 2

 C H A N N E L    PKTS/SEC  BYTES/SEC    C H A N N E L    PKTS/SEC  BYTES/SEC
    (01-16)                                 (17-32)
STDMSG                   0          0  TEMP_CHANNEL_10           0          0
STDDIST                  0          0                            0          0
STDARCH               1003      64239                            0          0
STDRESP                  0         53                            0          0
STDSTATUS                0          0                            0          0
TEMP_CHANNEL_01          0          0                            0          0
TEMP_CHANNEL_02          0          0                            0          0
TEMP_CHANNEL_03          0          0                            0          0
TEMP_CHANNEL_04          0          0                            0          0
TEMP_CHANNEL_05          0          0                            0          0
TEMP_CHANNEL_06          0          0                            0          0
TEMP_CHANNEL_07          0          0                            0          0
TEMP_CHANNEL_08          0          0                            0          0
DataChannel            999      55993                            0          0
CDTimeChannel            0        592                            0          0
TEMP_CHANNEL_09          0          0                            0          0

12:48:14.869                    TOTAL SYSTEM THROUGHPUT       2005     120879
```

Figure 4-1: ChanMan channel rates page

The channel rates screen gives an overview of all channels in the system. The channels 1-32 are listed on the first page. Channels 33-64 are listed on the second page and can be accessed by pressing the ‘l’ (lower case L, not one) key. The channel rates screen shows the packets/sec and bytes/sec for each channel and for the total system throughput. When examining this screen, here are several points to remember:

- If data is being archived (as is the case in most systems), STDARCH should have some data flowing through it.
- STDMSG is a low rate channel. If more than several packets a second flow through STDMSG, a process is repeatedly reporting an error; check the log files.
- Named channels that pass data between interface processes and data processing processes should have data flowing if the interface is active.
- STDARCH’s data rate should approximately be the sum of all channels feeding DataProc.

Pressing the ‘j’ and ‘k’ will show the channel details page. Figure 4-2 shows the channel details page for the STDARCH channel.



```
|Channel number  3                 S E R V E R  C L I E N T     Page 1 of 2 |
|Channel name    STDARCH                  OPEN          OPEN  state          |
|Channel type    STANDARD                   16             1  user_count     |
|                                        18847         18853  user[0]        |
| lock value       0                         0             0  signal_q_count |
|                                            0         18853  signal_q[0]    |
|   cdb_offset    0x1eb60                 6016          6016  cdb_offset     |
|     cdb_size      40000                    0             0  idle_count     |
|  cdb_max_pkt      19992                                                    |
|   free_bytes      40000                   17             1  opens          |
|                                            1             0  closes         |
|suspended FALSE  write_fault FALSE   4253890       4253890  success        |
|signaling FALSE  w_flt_state FALSE         0             0  bad_size       |
|                                   272276600     272276600  bytes          |
|next_avail_chan_num     00                 0       3955082  blockables     |
|next_avail_seq_num   00000                 0       3955082  blocks         |
|                                           0             0  lock_waits     |
|           SERVER     CLIENT               0             0  errors         |
| pkts/sec     1003       1003              0             0  sigqueue_eacces |
|bytes/sec    64239      64239              0             0  sigqueue_einvals|
|                                           0             0  sigqueue_eperms |
|12:48:28.130                               0             0  sigqueue_esrchs |
```

Figure 4-2: ChanMan channel details page

The channel details page provides significant information about an individual channel. The following key points should be remembered when examining this page:

- For a channel to transmit data, one client and at least one server must have the channel open. The state of the server and client are shown in the upper right of the screen.
- When a serious problem occurs in the system, data sometimes stop flowing between processes. This problem is characterized by packets/sec and byte/sec values of zero. Diagnose this problem by examining the free bytes field on the center portion of the left side of the screen:
  - If the value of free bytes is zero or near zero, then the client is not reading data from the channel.
  - If the value of free bytes is equal to the CDB size value (just above it), then the server is not writing data to the channel.
- When a channel is working properly, but inactive, you will typically see the client blockables and blocks field increase.
- If a lot of data is flowing through a channel, it is possible for the writer to send data faster than the reader can process it. This is typically characterized by a large number of blockables and blocks for the server.
- Finally, the total number of bytes processed by the channel is available for viewing.

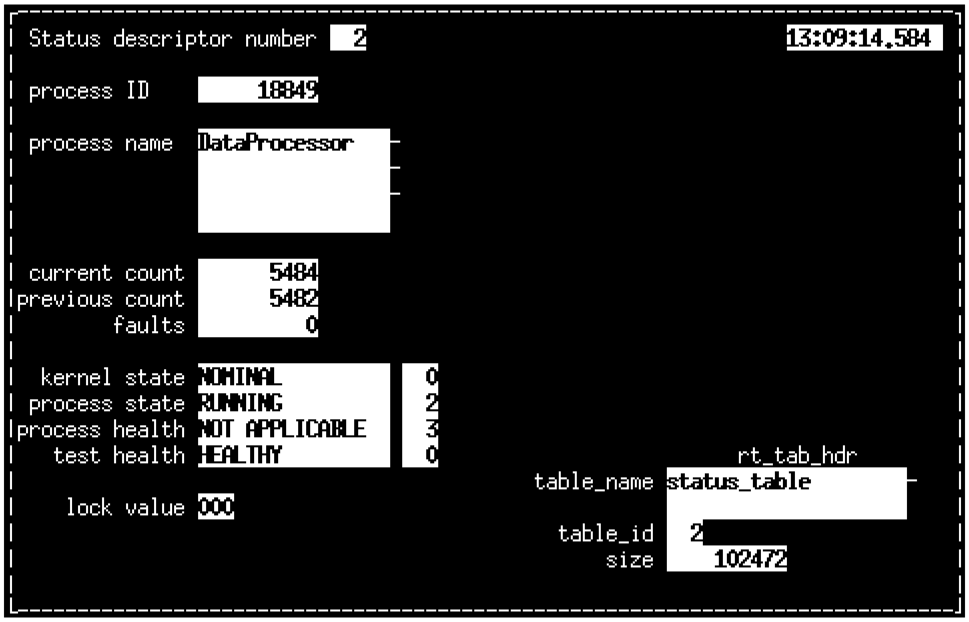
To exit *ChanMan*, use the ‘x’ key.

4.6.2 Process Status Monitoring

StatMon is a curses based utility that allows monitoring of both project and process health. A project must be up and running and the *KPATH* must be set before executing *StatMon*. *StatMon* can be executed from the UNIX command prompt:

```
% StatMon
```

To obtain a listing of the keystrokes used to navigate in *StatMon*, press the 'h' key. To make the help screen go away, press the space bar. Figure 4-3 displays a sample *StatMon* screen.



```
Status descriptor number  2                              13:09:14.584

process ID        18849

process name  DataProcessor




 current count        5484
previous count        5482
        faults           0

  kernel state  NOMINAL           0
 process state  RUNNING           2
process health  NOT APPLICABLE    3
   test health  HEALTHY           0                   rt_tab_hdr
                                          table_name status_table
    lock value  000
                                            table_id  2
                                                size    102472
```

Figure 4-3: Sample *StatMon* page

When viewing the status page, keep the following points in mind:

- If the process ID field is –1, then the descriptor is unused.
- Current count and previous count should be within one or two counts of one another. They should also increment at a rate of approximately once per second.
- Under normal circumstances, kernel state, process state, and project health should be identical to the above. Process health will be either "NOT APPLICABLE" or "HEALTHY". When a process's health is "NOT APPLICABLE", it does not provide health information to the system.

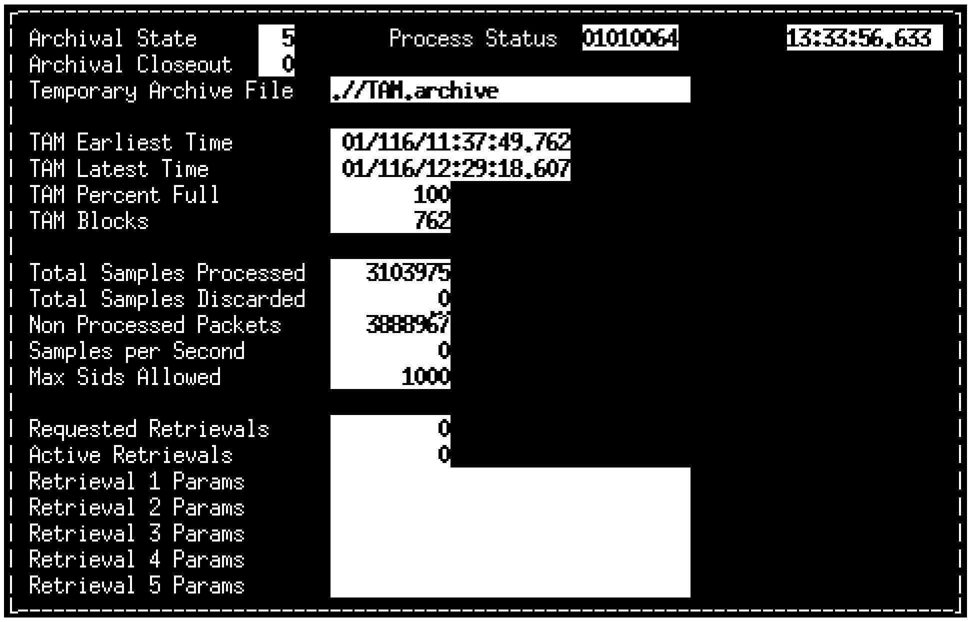
To exit *StatMon*, use the 'x' key.

4.6.3 Archive Monitoring

ArsMon is a curses based utility that allows monitoring of the archive subsystem. A project must be up and running and the *KPATH* must be set before executing *ArsMon*. *ArsMon* can be executed from the UNIX command prompt:

```
% ArsMon
```

To exit *ArsMon*, use the 'x' key. Figure 4-4 displays a sample *ArsMon* screen.



```
Archival State            5       Process Status  01010064          13:33:56.633
Archival Closeout         0
Temporary Archive File    .//TAM.archive

TAM Earliest Time         01/116/11:37:49.762
TAM Latest Time           01/116/12:29:18.607
TAM Percent Full                100
TAM Blocks                      762

Total Samples Processed     3103975
Total Samples Discarded           0
Non Processed Packets       3888967
Samples per Second                0
Max Sids Allowed               1000

Requested Retrievals              0
Active Retrievals                 0
Retrieval 1 Params
Retrieval 2 Params
Retrieval 3 Params
Retrieval 4 Params
Retrieval 5 Params
```

Figure 4-4: Sample ArsMon screenshot

4.6.4 Command Line Retrievals

Several utilities exist to perform retrievals from the command line within CCTK. To perform retrievals using a graphical interface, reference the *Retriever* section in the *CCTK User's Manual*. Like the other command line utilities discussed thus far, a project must be up and running and the *KPATH* must be set before executing these utilities.

NrtRetrieve is the back-end process that performs retrievals during project execution. Both *SimpleNrt* and *Retriever* use *NrtRetrieve* to perform retrievals. *NrtRetrieve* accepts a retrieval parameter file as input. This special file is based upon the Tcl script language and describes the retrieval to be performed. See *NrtRetrieve* on page 142.

To perform a retrieval using *NrtRetrieve* from the command line, use the following command template:

```
% NrtRetrieve [-o output_file] [-b start_time] [-e end_time] [-t timeout] \
      parameter_file [parameter file …]
```

The simplest way to generate a parameter file is to use *Retriever*. The saved file output of *Retriever* is a parameter file. It is also possible to generate a retrieval parameter file by hand, but it is not recommended.

Using *NrtRetrieve* and a retrieval parameter file gives the administrator full control over the retrieval. However, *SimpleNrt* can be used in some circumstances. *SimpleNrt* allows retrievals of FD's, but does not allow any filtering to be performed on the FD's. To perform a retrieval using *SimpleNrt*, use the following command template:

```
% SimpleNrt [-b start_time] [-e end_time] -o output_file fd [fd ...]
```

For more information on *NrtRetrieve* or *SimpleNrt*, see the manual pages in Section 8.

4.6.5 Batch Retrievals

NrtRetrieve, as described above, is capable of performing batch retrievals. Simply pass multiple retrieval parameter files to *NrtRetrieve* on the command line and it will perform each retrieval in turn. As discussed above, the simplest way to generate a retrieval parameter file is to use *Retriever*.

4.6.6 Grace Scripts

CCTK uses an application called Grace to support plotting of retrieved data. Information on Grace can be found at the Grace website (http://plasma-gate.weizmann.ac.il/Grace/). Grace is a flexible tool that can easily be interfaced with the system in a variety of ways. CCTK provides two scripts *RetGraceGenPlot* and *RetGraceParseFd* in order to facilitate integration with Grace. *RetGraceGenPlot*, a Tcl script, will execute Grace with the appropriate command line options to display a graph for a given delimited retrieval file. The manual page for *RetGraceGenPlot* can be found on page 152. *RetGraceParseFd* will parse a given FD from a given delimited retrieval file and send the data to standard output data. It can then be ingested directly into Grace using the pipe command line option. The manual page for *RetGraceParseFd* is on page 153. Using these scripts as an example, it is possible to create custom plots by configuring the command line options and configuration file options provided by Grace.

4.6.7 CCTKsh

CCTKsh is a scripting interface to CCTK based upon the Tcl scripting language. This section discusses the additions to Tcl that allow it to interact with CCTK, but does not discuss the Tcl scripting language. If you are unfamiliar with the Tcl scripting language, it would be beneficial to familiarize yourself with the language before reading this section.

To execute *CCTKsh*, simply type *CCTKsh* at the command prompt:

```
% CCTKsh
%
```

It is also possible to load the CCTK extensions to Tcl using the require command:

```
% CCTKsh
% package require cctk
0.1
%
```

As with the other system utilities, a project must be up and running and the *KPATH* must be set before executing *CCTKsh*.

As an example, it is possible to obtain the current value of a measurement using the ::cctk::GetMeasValue command. The following shows how to use this command to obtain the current value of the FD “Test Analog #1”:

```
% CCTKsh
% package require cctk
0.1
% ::cctk::GetMeasValue “Test Analog #1”
0.0
%
```

By the same token, it is easy to issue a CCTK command using the ::cctk::IssueCmd Tcl command. The following shows how to use this Tcl command to issue the CCTK command:

```
% CCTKsh
% package require cctk
0.1
% ::cctk::IssueCmd “Start_CDTime”
nak
%
```

In the above case, the command was not acknowledged (probably because no countdown time was set and thus, it was not possible to start it).

There are several Tcl commands that interface with the CCTK system.

Table 4-5 lists the valid *CCTKsh* commands and the arguments they accept.

Table 4-5: CCTKsh command descriptions

Command	Description
::cctk::GetMeasList	Returns the list of valid measurements for the project.
::cctk::GetMsgList	Returns the list of valid system messages for the project.
::cctk::GetCmdList	Returns the list of valid command for the project.
::cctk::GetDistVector [-raw] <fd>	Returns the distribution vector associated with the fd.
::cctk::GetMeasValue <fd>	Returns the current value of a measurement.
::cctk::SetMeasValue <fd> <new value>	Attempts to set the current value of a measurement.
::cctk::IssueCmd <fd>	Issues a command.
::cctk::IssueValue <fd> <parameter value>	Issues a command that has a single parameter.
::cctk::GetFdInfo <fd> [property]	Returns information about a particular property of a descriptor.
::cctk::SetFdInfo <fd> <property> <new value>	Sets a particular property of a descriptor.
::cctk::GetHealth	Returns the current health of the CCTK system.

After running *CCTKsh*, these commands can be typed at the prompt as shown above and the results will be displayed at the terminal. Many of the above commands make excellent debugging tools when troubleshooting the system.

It is also possible to create complex scripts using Tcl, Tk, and the CCTK extensions to the language. Table 4-6 lists several examples of Tcl scripts that are available for reference, their location, and a short description of each.

Table 4-6: Tcl scripts

Script Name	**Location**	**Description**
measurement_update.tcl	*${CCT_HOME}/projects/SIMPLE/bin*	Updates the description and units of an analog measurement in real-time while the system is running. Uses the ::cctk::SetFdInfo Tcl command.
simple_archive.tcl	*${CCT_HOME}/projects/SIMPLE/bin*	Reads the current value, description, and units of a measurement and writes the data to a file. Uses the ::cctk::GetFdInfo and ::cctk::GetMeasValue Tcl commands.
IgniterSafing.tcl	*${CCT_HOME}/projects/SuperLokiDemo/bin*	Issues a series of commands to open all of the firing circuits in the SuperLokiDemo. This script is executed when the emergency safing button is pressed on the firing circuit displays. It uses the ::cctk:IssueCmd Tcl command.

5 POST PROJECT EXECUTION

This section discusses tasks that can be performed after a project has been terminated. At this time, there are two primary tasks:

- Post Project Administration
- Historic Retrievals

These topics are covered in the following sections.

5.1 Post Project Administration

After a project is shutdown, there are a few simple tasks to perform. These tasks include:

- Permanent storage of project related files
- Clean-up of the project directory

Once a project is shutdown, it may be necessary to arrange for permanent storage of certain project related files. If a historical record of the project is needed, CCT suggests that the project directory be copied to a permanent storage location. By copying the project directory, a record of the project configuration, the configuration database, the project log files, the archive files, and the temporary files are all saved. These files are invaluable for future analysis of the project. CCT recommends that each project administrator develop a plan for storing project files. At a minimum, the archive files must be stored if historic retrievals need to be performed on the project in the future.

After the appropriate files have been copied to the permanent storage solution, the project directory may be cleaned up. This step is not absolutely necessary as at the start of the next project, all dynamic resources will automatically be removed and recreated. However, if it becomes necessary to manually clean the project directory, the following files may safely be deleted:

- *${KPATH}/tmp*: The project temporary directory
- *${KPATH}/TAM**: The project archive files

5.2 Historic Retrievals

To perform historic retrievals for a past project, the archive files associated with the project must be accessible. Storage of historic archive files is site dependent, as discussed in Section 5.1. Once the correct archive files have been located, it is simple to perform historic retrievals.

Historic retrievals work in a similar manner to near real-time retrievals discussed in Section 4.6.4. Graphical retrievals can be performed using *Retriever*. Reference the *CCTK User's Manual* for information on using *Retriever* to perform historic retrievals. Historic command line retrievals can be performed using the applications *HistRetrieve* or *SimpleHist*.

HistRetrieve is analogous to *NrtRetrieve*. *HistRetrieve* is the back-end process that performs retrievals during project execution. Both *SimpleHist* and *Retriever* use *HistRetrieve* to perform historic retrievals. *HistRetrieve* accepts a retrieval parameter file as input. This special file is based upon the Tcl script language and describes the retrieval to be performed. For information on the contents of a retrieval parameter file, see *HistRetrieve* on page 135.

To perform a historic retrieval using *HistRetrieve* from the command line, use the following command template:

```
% HistRetrieve [-o output_file] [-b start_time] [-e end_time] [-p path] \
       [-f filebase] [-s[s]] parameter_file [parameter file …]
```

HistRetrieve adds two important options (–p and –f) over *NrtRetrieve*. –p indicates the path where the historic archive files are located. If –p is not specified, then it defaults to the current working directory. –f indicates the base filename of the archive files. –f defaults to TAM if it is not specified.

The simplest way to generate a parameter file is to use *Retriever*. The saved file output of *Retriever* is a parameter file. It is also possible to generate a retrieval parameter file by hand, but it is not recommended.

Using *HistRetrieve* and a retrieval parameter file gives the administrator full control over the retrieval. However, *SimpleHist*, which is analogous to *SimpleNrt*, can be used in some circumstances. *SimpleHist* allows retrievals of FD's, but does not allow any filtering to be performed on the FD's. To perform a retrieval using *SimpleHist*, use the following command template:

```
% SimpleHist [-l path] [-f filebase] [-b start_time] \
       [-e end_time] -o output_file fd [fd ...]
```

Again, note the differences between *SimpleHist* and *SimpleNrt*. Both options –l, to specify the path, and –f, to specify the filename base, are present.

For more information on *HistRetrieve* or *SimpleHist*, see the manual pages in Section 8.

As with *NrtRetrieve*, *HistRetrieve* is capable of performing batch historic retrievals by placing multiple parameter files on the command line. By using the Grace scripts described in Section 4.6.6 as a start point, it is possible to create scripts that generate a series of plots from a historic archive file. Please contact CCT for further assistance with historic batch and historic batch plot retrievals.

6 SIMULATION

The CCTK simulation environment provides a generic tool for test and debug of system applications and configurations. It is also useful in creating training scenarios and test simulations necessary for operational readiness.

The simulator is accessed directly from the CCTK command line and supports three modes of operation:

- *dsim*: Debug simulator, outputs to a file or standard out
- *gsim*: GLG simulator, outputs to a GLG display
- *csim*: CCTK simulator, interacts with CCTK

This section discusses the general use of the three different simulators and the simulation language in general.

6.1 Running the Debug Simulator

The debug simulator can be run in two different modes: interactive or script driven. The simulator will act similar to your shell in a manner that you can interactively enter commands at a prompt or you can feed it a script file to process. To start the debug version of the simulator, simply type the following:

```
% dsim
```

You should be greeted with the simulator prompt:

```
sim>
```

Type the following command:

```
sim> Sim_Run
```

The *Sim_Run* command will cause the simulator to enter its processing loop. The simulator operates on a fixed cycle loop that defaults to 1 Hz. You should see the following repeated over and over until it is stopped by sending an interrupt signal. This can be done with the UNIX kill signal, but <Ctnl-c> at the terminal will do the same task. *Sim_Run* will output the following:

```
Update Cycle Started ...
Check For Commands Called...
Update Cycle Finished ...
Update Cycle Started ...
Check For Commands Called...
Update Cycle Finished ...
```

In this case, the debug output module prints a statement each time the simulator update cycle starts, commands are checked for, and the update cycle finishes. In addition, after commands are processed, the simulator processes events, and updates measurements. The simulator internally implements event and measurement processing while command processing is deferred to the external I/F module. It is important to understand the ordering of steps in the update loop:

- Update Cycle Started: Allows the external I/F module to perform processing prior to any other activities for this update cycle.
- Check for Commands: Allows the external I/F module to check for any outstanding commands and notify the simulator of those commands.
- Process Events: The simulator will internally process events and performs the actions necessary to satisfy the event.
- Process Measurements: The simulator will update each measurement based upon its algorithm. The external I/F module will be notified if any item changes.
- Update Cycle Ended: Allows the external I/F module to perform processing at the end of the update cycle. As an example, the glg module uses this to sync the remote screen.
- Sleep: Sleep till the start of the next update cycle.
- Update Cycle Finished: Defers to external I/F module for processing.

6.2 Running The GLG Simulator

The GLG option allows a client/server implementation where data can be fed to a GLG display via a remote application. The simulator uses this feature of GLG to provide simulation to GLG displays. For the GLG simulator to work, the following properties must be set in the viewport of the GLG object:

- ServerEnabled
- ServerName

Please see the GLG documentation for more information on these resources. The *GLGbuilder* viewport is a server with the name <GlgDrawingAreaServer> so it is possible to use the simulator to simulate data into the *GLGbuilder*. Once the "ServerName" is known, the GLG simulator can be started with the following command:

```
% gsim -s <server_name> [script_name]
```

You must specify the <server_name> on the command line. Just like with the debug simulator however, the script_name is optional. In fact, all of the simulator commands that work in debug will work with the GLG simulator. To manipulate GLG data from the simulator, name the simulator measurement the same as the GLG resource name.

6.3 CCTK Simulation

The CCTK version of the simulator is called *csim*. The *csim* simulator accepts the command line arguments shown in Table 6-1.

Table 6-1: csim command line argument descriptions

Argument	Description
<-o output_channel>	A required argument that specifies a CCTK channel name. All linked data packets will be written to this channel. Only one may be specified.
[-i input_channel]	An optional argument that specifies a CCTK channel name. Multiple input channels may be specified. Commands will be read from each channel once per cycle.
[-p port_fd]	An optional argument that specifies a CCTK port fd. None are required but multiple may be specified. The input channel will be registered with each port.
[-k kpath]	An optional argument that specifies the kpath. If kpath is omitted, then the KPATH environment variable must be set.
[script_file ...]	Used to specify a set of script files to process. If no file is specified, an interactive shell will be used.

The output channel must be specified and must exist. This will usually point to a *DataProc* that will then process the linked data packets produced by the simulator.

The CCTK version of the simulator handles commanding. The simulator can receive commands in one of two ways, it can receive commands over a hard-coded named channel. These are specified by the -i option on the command line. You may specify as many channels as necessary. The simulator can also receive commands by registering in the port tables for commands for a specified port.

In this case, a series of port_fds must be specified. Once each cycle, the simulator will check for commands on each input channel and on the unnamed channel assigned to the port FD’s. When a command is received, the proper simulation command will be issued.

To register a command with the simulator, use the *Sim_NewCmd* command. The following registers a command that will stop the simulator whenever the CCTK *StopSim* command is received.

```
sim> Sim_NewCmd -type stop -name StopSim
```

There are two key things to remember when using the CCTK simulator:

- The simulator DOES NOT perform reverse conversion on any of the data so you must be simulating data that is either not converted or you must simulate the raw value of the measurement.
- The simulator DOES NOT pass parameters to commands. When a command is received, only the Fd of the command is passed from the CCTK interface module to the simulator.

6.4 Simulation Engine and Tcl

The simulation language is based upon Tcl, so all valid Tcl commands and syntax are valid within the simulator. This greatly extends and enhances the capabilities of the simulator. Please see a Tcl reference book or the World Wide Web for more information on programming in Tcl.

Although it is possible to build simulation scripts without an understanding of the Tcl language, the full power of the simulator can only be obtained when Tcl constructs are used in creating simulation scripts.

6.5 Simulation Language

This section focuses on the simulation language. The simulation language is the same for all three versions of the simulator (*dsim*, *gsim*, *and csim*). In fact, many times, the scripts tested using *dsim* can easily and quickly be converted to csim scripts with little or no work.

6.5.1 Key Simulation Modules

All simulation scripts are composed of three key components:

- Measurements Modules
- Command Modules
- Event Modules

The simulator provides a series of commands to create, query, update, and delete these three types of modules.

Measurement modules are used to update data within the simulator. A measurement module can be responsible for updating a single piece of data or a hundred pieces of data. All simulation scripts will be composed of one or more measurement modules. The simulator defines many different kinds of measurement modules (such as constant, random, and file). Details on the different kinds of measurement modules are given below.

Command modules are used to respond to commands entering the simulator. The simulator provides a means by which it can be commanded. Whenever a particular command is received, the simulator sees if a command module has been defined to handle that command. If a command module has been defined, it will be executed. Just as there are many different types of measurement modules, there are many different types of command modules. Details on the different kinds of command modules are given below.

Event modules are used to cause asynchronous events to occur based upon some condition. For example, an event module can be used to change the parameters of a measurement module when a specific CDT is reached. Another example would be when a measurement exceeds a value, a special script could be evaluated. Just as with commands and measurements, there are many different types of event modules. Details on the different kinds of event modules are given below.

6.5.2 Update Cycle

The simulator operates on a fixed update cycle. Once each update cycle it performs the following tasks:

- Processes any pending commands.
- Processes all event modules, executing those whose condition is true.
- Updates all measurement modules.

The update cycle is started by issuing the <Sim_Run> command to the simulator. Once the <Sim_Run> command is issued, the simulator will start the update cycle and loop until it is stopped by either an OS signal (INTR/TERM/KILL) or by having the <Sim_Stop> command issued. The simulator can also be programmed to run a fixed number of iterations and then stop. The following are the details on the <Sim_Run> and <Sim_Stop> commands:

```
sim> Sim_Run [update_rate [iterations]]
```

Sim_Run will cause the simulator to enter its update loop. If no update rate is specified, the default of once per second will be used. If an update rate is given, the simulator will perform its update loop once each number of seconds specified. If the iterations is set, the simulator will loop that number of iterations before stopping.

```
sim> Sim_Stop
```

Sim_Stop will cause the simulator to exit its update loop at the beginning of the next cycle.

6.5.3 Core Simulator Commands

The simulator defines four commands for each type of module. The commands for operating on the measurement modules are:

- Sim_NewMeas
- Sim_UpdateMeas
- Sim_QueryMeas
- Sim_DelMeas

The commands for the command modules are:

- Sim_NewCmd

- Sim_UpdateCmd
- Sim_QueryCmd
- Sim_DelCmd

The commands for the event modules are:

- Sim_NewEvent
- Sim_UpdateEvent
- Sim_QueryEvent
- Sim_DelEvent

In addition, there are several other miscellaneous commands which are described in this section.

All of the above commands take a series of arguments. These always take on the following form:

```
<argument=value>
```

Where argument is the argument that needs to be set and value is the value it needs to be set to. Each of the above commands can take multiple argument/value pairs. The order of the pairs is unimportant. However, some arguments are only valid for certain modules and/or certain types of modules. Details on which commands accept which arguments are given below. Details on arguments defined by different types of modules are provided in modules Section 6.5.4 below.

All of the above commands are valid Tcl commands. They will behave as normal Tcl commands. If an error occurs while processing, a Tcl error will be returned and a textual description of the error will be placed in the results buffer. The textual description should provide adequate information for determining the source of the problem.

6.5.3.1 New Simulation Command Definition

```
Sim_New{Meas|Cmd|Event} <name=module_name> <type=module_type>
[<active={true|false}] [<argument=value> <...>]
```

Sim_New{Meas|Cmd|Event} is used to create a new module. To create a new module, at least two arguments are required. The first is the name of the module. This name will be used as a reference to the module in other parts of the simulation script. The second argument is the type of module to create. As stated previously, there are many different types of measurement, command, and event modules, this type indicates what needs to be created. Based upon the type, additional arguments may be required and additional optional arguments may be specified. All Sim_New{Meas|Cmd|Event} commands can take at least one optional argument. That option argument defines whether or not the module is active. An inactive module will perform/take no action. The default is true.

For example, the following command creates a measurement module named “TestMeas” of type “const” and sets its value to “10”.

```
sim> Sim_NewMeas name=TestMeas type=const value=10
```

6.5.3.2 Update Simulation Command Definition

```
Sim_Update{Meas|Cmd|Event} <name=module_name> [<active={true|false}]
[<argument=value> <...>]
```

Sim_Update{Meas|Cmd|Event} is very similar to the new command. The update command is used to change any argument associated with a particular module. (NOTE: it may not be possible to change some arguments, for example, once a type is specified, it cannot be changed). Any module will always be able to accept active as part of the update arguments. Other arguments are defined by a particular module.

For example, the following command updates the measurement module created in the new example:

```
sim> Sim_UpdateMeas name=TestMeas value=20
```

To set it to inactive, the following command is issued:

```
sim> Sim_UpdateMeas name=TestMeas active=false
```

6.5.3.3 Query Simulation Command Definition

```
Sim_Query{Meas|Cmd|Event} <name=module_name> <query=argument>
```

Sim_Query{Meas|Cmd|Event} allows individual arguments associated with a module to be queried. The query command always takes two arguments, the name of the module to query and the argument to query. Any argument known to a module can be queried, however, only one argument can be queried at a time. The argument is returned as a string.

For example, the following command queries the value of the measurement module created in the previous example:

```
sim> Sim_QueryMeas name=TestMeas query=value
```

The active status can be checked as well:

```
sim> Sim_QueryMeas name=TestMeas query=active
```

6.5.3.4 Delete Simulation Command Definition

```
Sim_Del{Meas|Cmd|Event} <name=module_name>
```

This command allows a module to be removed from the simulator. The delete command always takes a single argument that specifies the name of the module to delete.

For example, to remove the module created previously:

```
sim> Sim_DelMeas name=TestMeas
```

6.5.3.5 Issue Simulation Command Definition

```
Sim_Issue <command name>
```

This command will cause the specified command module to execute. This command is provided as a way to force a command to execute, bypassing the normal flow through the external interface module.

6.5.3.6 Time Simulation Command Definition

```
Sim_Time <time arguments>
```

The simulator has access to the system time facilities. Depending on which variant of the simulator is currently being executed, the backend that drives the time in the simulator is different. However, the front-end that the user interfaces with is always the same. The following set of commands should always work from within the simulator.

```
sim> Sim_Time cdt
```

Queries the current state of the countdown time. A list of two arguments is returned. The first is the state of the clock, running or stopped. The second is the number of microseconds the clock is currently set to. If the clock is running, this will change with each query of the clock.

```
sim> Sim_Time cdt start
```

Starts the countdown (cdt) clock running.

```
sim> Sim_Time cdt stop
```

Stops the cdt clock from running.

```
sim> Sim_Time cdt set <time>
```

Sets the cdt clock to the specified time. If the clock is running, it continues to run, but the new time is used. If the clock is not running, it will remain stopped until started. <time> should be the number of microseconds you wish to set the clock to. It can be positive or negative.

```
sim> Sim_Time sutc
```

Queries the current state of the simulated universal time (SUTC). A list of two arguments is returned. The first is the state of the clock, running or stopped. The second is the number of microseconds the clock is currently set to. If the clock is running, this will change with each query of the clock.

```
sim> Sim_Time sutc start
```

Starts the sutc clock running.

```
sim> Sim_Time sutc stop
```

Stops the sutc clock from running.

```
sim> Sim_Time sutc set <time>
```

Sets the sutc clock to the specified time. If the clock is running, it continues to run, but the new time is used. If the clock is not running, it will remain stopped until started. <time> should be the number of microseconds you wish to set the clock to. It must be positive.

```
sim> Sim_Time utc
```

Returns the current, real-utc.

6.5.4 Core Simulator Modules

6.5.4.1 External Interface Modules

It is possible that external interface modules can implement simulator commands for specific purposes. This section describes the interface specific commands.

```
Sim_GlgIf triplet new <resource> <{meas1 meas2 meas3}>
```

This command allows the simulator to simulate a GLG triplet. You specify three measurements that are being simulated in the normal manner. At the end of each cycle, GLG will export these measurements as the specified resource. Note, this command is only valid in the GLG simulator (gsim).

6.5.4.2 Simulation Measurement Modules

This section discusses the different types of measurement modules defined within the core simulation environment and the arguments that the modules accept.

type=const

Constant measurement module's values do not change with time. They can only be changed by performing an update on the module. Each constant module defines a single measurement within the simulation environment. The constant type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). Due to the implementation of SimEngine, setting force to false for a constant, will cause NO DATA to be sent at any time. Therefore, it is ill advised to set force to false when using the const type. (default = true)
- value – the constant value. (default = (uninitialized))

type=rand

The random type will generate a random number within a range each cycle. Each random module defines a single measurement within the simulation environment. The random type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)
- value – the current value.
- lower – this represents the lower bound. (default = 0)
- upper – this represents the upper bound. (default = 1)

- integral – this is a true/false argument whether or not the random number should be an integer type or a floating point type. (default = false)

type=square

This type will allow a square wave to be generated. Multiple arguments can be passed to the procedure as defined below. The square type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)
- value – the current value.
- period – This indicates the number of seconds to complete one iteration of the wave. (default = 10.0)
- upper – a string to return when the wave is at its upper value. (default = upper)
- lower – a string to return when the wave is at its lower value. (default = lower)

type=sine

This type will allow several different types of sine waves to be produced. Multiple arguments can be passed to the procedure as defined below. The sine type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)
- value – the current value.
- amplitude – This indicates the maximum "height" of the wave. (default = 1)
- offset – This is the offset from zero. (default = 10.0)
- period – This indicates the number of seconds to complete one iteration of the wave. (default = 10.0)
- bias – Bias can be positive, negative, or 0. A 0 or neutral bias will generate a "normal" sine wave. A positive bias will cause the lower part of the wave form to flip. This is similar to taking the abs() of the result of the sine wave. A negative bias will cause the

opposite to happen and the upper part of the wave form will flip. This is similar to taking the -abs() of the result. (default = 0)

- integral– this is a true/false argument whether or not the random number should be an integer type or a floating point type. (default = false)

type=saw

This type will allow several different types of sawtooth waves to be produced. Multiple arguments can be passed to the procedure as defined below. The saw type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)
- value – the current value.
- amplitude – This indicates the maximum "height" of the wave. (default = 1)
- offset – This is the offset from zero. (default = 10.0)
- period – This indicates the number of seconds to complete one iteration of the wave. (default = 10.0)
- integral– this is a true/false argument whether or not the random number should be an integer type or a floating point type. (default = false)

type=ramp

This type will allow a value to "ramp" from some initial value to some final value. Several different types of ramping is supported. These types are described below:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)
- value – the current value.
- initial – the initial value for this algorithm. This is where the ramping will start. (required argument)
- final – the final value for this algorithm. This is where the ramping will end. (required argument)

- period – the time that it will take to go from the initial value to the final value. (default = 10.0)
- delay – if this time is set the ramping will not start until after the delayed time is complete. This can be used to implement a pause before the value actually starts to change. (default = 0)
- algo – this indicates the algorithm to use during the ramping. Currently, only a linear algorithm is supported. In the future other types may be supported. (default = linear)
- integral– this is a true/false argument whether or not the random number should be an integer type or a floating point type. (default = false)

The ramp command will reset itself if ANY modifications are made to the above parameters using the query command.

type=tcl

This is a special type that will obtain the next value of a measurement from a Tcl script. Each Tcl module defines a single measurement within the simulation environment. Each time an update cycle is performed, the specified Tcl script will be evaluated. The new value should be returned from this script. The Tcl type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)
- value – the current value.
- script – the script to evaluate. <this is required>

type=delim

This type will read its input from a file. The file can have any number of columns, delimited by a fixed character. Each update cycle, one row from the file will be read and fed into the simulator. Each column can either have a tag assigned to it or be ignored. The delim type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)

- delim – this indicates the delimiter character that is between the fields. (default = ,)
- ignore – this indicates the number of initial rows to ignore (probably because of headers). (default = 0)
- tags – this is a Tcl list. There should be the same number of items in the list as there are fields in the file. This identifies the tag to be associated with each field in the file. If a member of this list is blank, that field will be ignored. <this is a required parameter>
- repeat – this indicates the action to take when the end of the file is reached. If this is true, the simulator will continue reading from the beginning of the file. If this is false, all values will be held at their current state. (default = true)
- file – the full path to the file to read. This is a required argument.

type=cdt

This type will read its input from a file. The file can have any number of columns, delimited by a fixed character. Each update cycle, the "time" column will be checked against the current CDT. If the time has passed, then that row will be fed through the simulator. Each column can either have a tag assigned to it or be ignored. The delim type supports the following arguments:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- external – the external tag that the external interface uses to identify this measurement. If this is not defined, the name of the module is used. (default = not defined)
- force – force indicates whether or not the value should be sent updated only when it changes (force = false) or every cycle regardless of whether or not it changes (force = true). (default = false)
- delim – this indicates the delimiter character that is between the fields. (default = ,)
- ignore – this indicates the number of initial rows to ignore (probably because of headers). (default = 0)
- time – this is a number indicating which column has the time field. Assume the first column is 1. This number must be less than the maximum number of columns in the file. (default = 1)
- tags – this is a Tcl list. There should be the same number of items in the list as there are fields in the file. This identifies the tag to be associated with each field in the file. If a member of this list is blank, that field will be ignored. <this is a required parameter>
- file – the full path to the file to read. This is a required argument.

6.5.4.3 Simulation Command Modules

This section discusses the different types of command modules defined within the core simulation environment and the arguments that the modules accept.

type=tcl

This command type will evaluate the specified Tcl procedure whenever the command is received. The following arguments are supported:

- name – the name of the module.
- external – the external tag that the external interface uses to identify this command. If this is not defined, the name of the module is used. (default = not defined)
- active – whether or not the module is active. (default = true)
- proc – the procedure to evaluate. <this is required>
- script – the script to evaluate. <this is required>
- file – the file to evaluate. <this is required>
- args – this defines any additional arguments that will be passed to the procedure. (default = none defined)

NOTE: Only one of proc/script/file may be specified. Only the procedure will use the args/current parameters.

type=stop

This command will cause the simulator update cycle to stop when received. The following arguments are supported:

- name – the name of the module.
- external – the external tag that the external interface uses to identify this command. If this is not defined, the name of the module is used. (default = not defined)
- active – whether or not the module is active. (default = true)

6.5.4.4 Event Modules

This section discusses the different types of event modules defined within the core simulation environment and the arguments that the modules accept.

type=cdt

This event type will cause a Tcl procedure/script/file to evaluate whenever the cdt time passes. The following arguments are supported:

- name – the name of the module.
- active – whether or not the module is active. (default = true)
- cdt – the cdt time to examine.
- script – the script to evaluate. <this is required>
- args – this defines any additional arguments that will be passed to the Tcl code if a procedure/script is defined. (default = none defined)

The Tcl code will only be evaluated once. This will happen at the earliest cycle after the specified countdown time passes. Only one of proc/script/file may be specified.

6.6 Simulation Examples

Table 6-2 lists several example simulation scripts available for reference, their location, and a short description of each.

Table 6-2: Simulation scripts

Script Name	Location	Description
simple.sim	*${CCT_HOME}/projects/SIMPLE/sim*	A simple simulation script that simulates many of the measurements associated with the SIMPLE project. This simple simulation attempts to use most of the basic simulation module types.
derived.sim	*${CCT_HOME}/projects/SIMPLE/sim*	A more complicated script that derives the simulated value of a measurement using a Tcl script. Some knowledge of Tcl is helpful in understanding this script.
cdt.sim	*${CCT_HOME}/projects/SIMPLE/sim*	This script alters the simulation based upon the current CDT. Five analog measurements are simulated and their values are altered as the CDT counts down.
simple.sim	*${CCT_HOME}/projects/SuperLokiDemo/sim*	This simulation script is used to demonstrate SuperLoki Launch Operations. This script reacts to the commands present on the firing circuit display and modifies the measurements as appropriate.

7 RETRIEVAL PARAMETER FILE

The retrieval parameter file defines all of the parameters (such as FDS, times, filters, etc.) associated with a CCTK retrieval. This chapter defines the syntax of the retrieval parameter file, which *NrtRetrieve* and *HistRetrieve* take as input. The retrieval parameter file should rarely have to create this file by hand. The syntax of the file is provided here as a reference for assistance in resolving problems associated with retrievals.

7.1 File Syntax

Retrieval parameter files are simple text files, parsed by a Tcl interpreter, that describe what needs to be retrieved. All Tcl commands are valid. Additionally, a set of retrieval-specific command indicate exactly what to retrieve.

```
RetrievalType <PROCESSED | RAW | RAW_AND_PROCESSED>
OutputFile <ret_outfile>
[StartTime YYYY/DDD/HH:MM:SS.mmm]
[StopTime YYYY/DDD/HH:MM:SS.mmm]
[Duration HH:MM:SS.mmm]
[SystemMsgs <YES/NO>]
Meas <Fd> <ANALOG_TYPE | UNSIGNED_INT_TYPE | SIGNED_INT_TYPE | \
DISCRETE_TYPE | BYTE_ARRAY_TYPE> <ALL | NORMAL | EXCEPTION | \
INVALID> \ <ALL_VALUES | ANY_CHANGES | DELTA_VALUE |
SPECIFIC_VALUE | INSIDE_RANGE | OUTSIDE_RANGE> \
[parameters ...]
Cmd <Fd> <CMD_DES_TYPE_S> <SYS_CMD | EI_CMD>
```

7.2 Command Descriptions

RetrievalType PROCESSED|RAW|RAW_AND_PROC

This keyword is used to specify the type of retrieval. The valid types are listed above. If RetrievalType is not specified, it defaults to PROCESSED.

OutputFile filename

This keyword specifies the name of the output file to use when storing the results of the retrieval. It can be overridden with command line options for *NrtRetrieve* and *HistRetrieve*. It must be specified either on the command line or in the file.

StartTime YYYY/DDD/HH:MM:SS.mmm

Indicates the time to start searching for Retrieval data. If this value is invalid or not present, ARS will default to the earliest time stored within the archive. Only two of StartTime, StopTime, and Duration may be specified; the third will be calculated automatically. This value can be overridden on the command line of *NrtRetrieve* and *HistRetrieve*.

StopTime YYYY/DDD/HH:MM:SS.mmm

Indicates the time to stop searching for Retrieval data. If this value is invalid or not present, ARS will default to the latest time stored within the archive. Only two of StartTime, StopTime, and Duration may be specified; the third will be calculated automatically. This value can be overridden on the command line of *NrtRetrieve* and *HistRetrieve*.

Duration HH:MM:SS.mmm

Indicates the duration of the retrieval. If this value is invalid or not present, ARS will default to the entire archive. Only two of StartTime, StopTime, and Duration may be specified; the third will be calculated automatically.

SystemMessages YES|NO

Indicates whether or not system messages should be retrieved. At this time, system messages can only be retrieved as a group. The default value is NO.

Meas FD MEAS_TYPE RETR_TYPE FILTER [FILTER OPTIONS ...]

Indicates that a measurement with the specified FD should be retrieved. The user must supply the measurement type so that some basic checks can be performed. The user must also supply the retrieval type, the retrieval type must be one of:

ALL, NORMAL, EXCEPTION, INVALID

The user must also specify a filter. Valid filters are: ALL_VALUES, ANY_CHANGES, DELTA_VALUE, SPECIFIC_VALUE, INSIDE_RANGE, and OUTSIDE_RANGE. Depending on the filter, one or two filter options may be necessary. ANY_CHANGES, DELTA_VALUE, SPECIFIC_VALUE all take one filter option; INSIDE_RANGE, OUTSIDE_RANGE take two filter options.

Cmd FD TYPE CMD_TYPE

Indicates that a command with the specified FD should be retrieved. The type will also be CMD_DES_TYPE_S currently. CMD_TYPE should either be SYS_CMD or EI_CMD.

7.3 Example

```
RetrievalType PROCESSED
OutputFile    ret_outfile
ParameterFile ret_paramfile
StartTime     99/152/16:04:00.123
StopTime      99/152/17:04:00.000
Duration      5
SystemMsgs    YES

Meas   MyFd1  BYTE_ARRAY_TYPE     NORMAL       ALL_VALUES
Meas   MyFd2  DISCRETE_TYPE       NORMAL       ANY_CHANGES
Meas   MyFd3  SIGNED_INT_TYPE     NORMAL       DELTA_VALUE   5
Meas   MyFd4  UNSIGNED_INT_TYPE   NORMAL       SPECIFIC_VALUE  50
Meas   MyFd5  ANALOG_TYPE         NORMAL       INSIDE_RANGE 1.0    10.0
Meas   MyFd6  ANALOG_TYPE         NORMAL       OUTSIDE_RANGE 10.0   20.0

Meas   MyFd1  BYTE_ARRAY_TYPE     EXCEPTION    ALL_VALUES
Meas   MyFd2  DISCRETE_TYPE       EXCEPTION    ALL_VALUES
Meas   MyFd3  SIGNED_INT_TYPE     EXCEPTION    ALL_VALUES
Meas   MyFd4  UNSIGNED_INT_TYPE   EXCEPTION    ALL_VALUES
Meas   MyFd5  ANALOG_TYPE         EXCEPTION    ALL_VALUES
Meas   MyFd6  ANALOG_TYPE         EXCEPTION    ALL_VALUES

Meas   MyFd1  BYTE_ARRAY_TYPE     INVALID             ALL_VALUES
Meas   MyFd2  DISCRETE_TYPE       INVALID             ALL_VALUES
Meas   MyFd3  SIGNED_INT_TYPE     INVALID             ALL_VALUES
Meas   MyFd4  UNSIGNED_INT_TYPE   INVALID             ALL_VALUES
Meas   MyFd5  ANALOG_TYPE         INVALID             ALL_VALUES
Meas   MyFd6  ANALOG_TYPE         INVALID             ALL_VALUES

Meas   MyFd1  BYTE_ARRAY_TYPE     ALL          ALL_VALUES
Meas   MyFd2  DISCRETE_TYPE       ALL          ANY_CHANGES
Meas   MyFd3  SIGNED_INT_TYPE     ALL          DELTA_VALUE   5
Meas   MyFd4  UNSIGNED_INT_TYPE   ALL          SPECIFIC_VALUE  50
Meas   MyFd5  ANALOG_TYPE         ALL          INSIDE_RANGE 1.0    10.0
Meas   MyFd6  ANALOG_TYPE         ALL          OUTSIDE_RANGE 10.0   20.0

Cmd    MyCmd1 CMD_DES_TYPE_S             SYS_CMD
Cmd    MyCmd2 CMD_DES_TYPE_S             EI_CMD
```

8 PROCESS MANUAL PAGES

This chapter includes process manual pages for key CCTK processes.

NAME

ArchiveControl - Archive monitor and control application

SYNOPSIS

ArchiveControl [-c command_channel] [-k kpath]

DESCRIPTION

Archive Control handles system commands to activate/inhibit the temporary archive (TAM) capability. Upon receipt of a valid command, Archive Control sets the appropriate state variable in a shared memory segment used by the ARS processes (TamArs) to control the state (active/inhibited) of the TAM archival.

This process must be started after the TamArs process has initialized, as it attaches to the shared memory segment created by TamArs.

OPTIONS

[-c command_channel]

This parameter is optional. Commands to Archive Control are sent via this channel. Commands are used to activate/inhibit archiving of data to the temporary and permanent archives. If this parameter is not specified, it defaults to NEXT_AVAIL_CHAN.

[-k kpath]

This parameter is optional. The Archive Control process must have a kpath to assist in accessing channels and shared memory tables. If not specified on the command line, the value is obtained from an environment variable named KPATH.

RETURN VALUES

If an error occurs during initialization, an error message is output and Archive Control exits. Otherwise, Archive Control runs until a terminate signal is received.

SEE ALSO

TamArs, PamArchive

NAME

ArsMon - monitor ARS process status

SYNOPSIS

ArsMon [kpath]

DESCRIPTION

ArsMon is an interactive display of the current status of the ARS. ArsMon must have a kpath to assist in accessing the status shared memory segment. The kpath may be specified on the command line or by an environment variable named KPATH. ArsMon has a help screen which lists the options the user may enter to control the ArsMon display. Hit the "h" key to display the help screen.

RETURN VALUES

Upon normal termination, ArsMon returns zero. Otherwise, an error message is output and one is returned.

NAME

CDTimeControl - The Count Down Time (CDT) Control process.

SYNOPSIS

CDTimeControl -p Port_FD_Name -o Output_Chan_Name [-muikcCsgGDh]

DESCRIPTION

The CDTimeControl process generates the system Count Down Time and sends it to DataProc.

CDTimeControl processes the following system commands to control the system CDT:

Set CDT - Set the initial CDT in seconds and milliseconds.

Start CDT - Start decrementing the current value of the CDT.

Stop CDT - Stop decrementing the current value of the CDT.

Cancel CDT - Cancel the current value of the CDT.

OPTIONS

-p Port_FD_Name

The Port_FD_Name is a required text string command line argument. The Port FD is used to identify the port descriptor the CDTimeControl process will use to set its distribution vector.

-o Output_Chan_Name
The Output_Chan_Name is a required text string command line argument. The Output Channel is the channel where the current CDT data packets are distributed.

-m Mode (<internal>/external)
The Mode is an option text string command line argument. The mode identifies whether the process time source is internal (default) or external. The external interface option is used to activate any extended mode interface built on top of the CDTimeControl process.

-u Update_Rate
The Update_Rate is an optional integer command line argument. The Update Rate specifies the number of milliseconds between successive CDT updates to the Output Channel. The Update Rate does not affect the CDT value. A default Update_Rate of 100 milliseconds will be used if one is not specified on the command line. The integer value of the Update Rate must be within the following range:
0 < Update_Rate < 1000 msec

-i Input_Chan_Name
The Input_Chan_Name is an optional text string command line argument. The Input Channel is the channel where CDTimeControl receives system commands. The next available Input Channel will be used as a default if one is not specified on the command line.

-k KPATH_Name
The KPATH_Name is an optional text string command line argument. The KPATH Name specifies the value to be used to set the KPATH environment variable. The KPATH environment variable is a required environment variable. The existing KPATH environment variable will be used as a default if one is not specified on the command line.

-c CDT_Value_FD_Name
The CDT_Value_FD_Name is an optional text string command line argument. The CDT Value FD specifies the FD where the current CDT seconds and milliseconds value will be distributed. The CDT Value FD

'CURRENT_CDTIME_TIME_VAL_FD' is specified in a system header file and will be used as a default if one is not specified on the command line.

-C CDT_Text_FD_Name
The CDT_Text_FD_Name is an optional text string command line argument. The CDT Text FD specifies the FD where the current CDT value will be distributed as a Julian Time of Year (JTOY) text string (-DDD:HH:MM:SS.mmm). The CDT Text FD 'CURRENT_CDTIME_TIME_TXT_FD' is specified in a system header file and will be used as a default if one is not specified on the command line.

-s CDT_Status_FD_Name
The CDT_Status_FD_Name is an optional text string command line argument. The CDT Status FD specifies the FD where the current CDT status will be distributed as a unsigned integer value. The CDT Status FD 'CURRENT_CDTIME_STATUS_FD' is specified in a system header file and will be used as a default if one is not specified on the command line.

-g GMT_Value_FD_Name
The GMT_Value_FD_Name is an optional text string command line argument. The GMT Value FD specifies the FD where the current GMT seconds and milliseconds value will be distributed. The GMT Value FD 'CURRENT_GMTIME_TIME_VAL_FD' is specified in a system header file and will be used as a default if one is not specified on the command line.

-G GMT_Text_FD_Name
The GMT_Text_FD_Name is an optional text string command line argument. The GMT Text FD specifies the FD where the current GMT value will be distributed as a Julian Time of Year (JTOY) text string (YYYY:DDD:HH:MM:SS.mmm). The GMT Text FD 'CURRENT_GMTIME_TIME_TXT_FD' is specified in a system header file and will be used as a default if one is not specified on the command line.

-D CDT_Dbg_Level
The -D option will enable debug mode. Debug mode causes verbose messages to be printed to a debug file in the log directory. The debug filename is DEBUG.CDTimeControl.### where ### is the CDTimeControl process ID. The CDT_Dbg_Level is an optional integer between 0 and 5 that determines how verbose the messages will be. Level 5 is most verbose.

-h
The -h option will print a help message to the screen. The help message indicates the current default values of the optional command line arguments.

MEASUREMENTS

The CDTime value is distributed by the CDTimeControl process using the three measurements listed below. One BYTE_ARRAY measurement contains the current signed CDTime value in seconds and milliseconds formatted as a sys_timeval_t structure. The other BYTE_ARRAY measurement contains the current CDTime value formatted as a signed Julian Time of Year text string that includes milliseconds (SDDD:HH:MM:SS.mmm). The UNSIGNED_INTEGER measurement contains a state code number indicating the operational state of the CDTime. The state codes are listed below. The GMTime value is distributed by the CDTimeControl process using the two measurements listed below. These are formatted similar to the CDTime except that the GMTime is unsigned (YYYY:DDD:HH:MM:SS.mmm). These FD’s and state codes are defined in a system header file (see KConstants.h). The CDTime measurement FD’s are updated at the update rate specified for the CDTimeControl process.

```
Measurement FD Name    Meas_Type       Meas_Size
-----------            -----           -----
CDTime_Time_Value      BYTE_ARRAY       8 bytes
CDTime_Time_Text       BYTE_ARRAY      18 bytes
CDTime_Status          INTEGER          4 bytes

GMTime_Time_Value      BYTE_ARRAY       8 bytes
GMTime_Time_Text       BYTE_ARRAY      22 bytes
```

CDT State Definition	Value	Description
CDTIME_STATUS_INVALID	0	CDTimeControl process not running.
CDTIME_STATUS_CANCELLED	1	CDTime is cancelled (CDT not set).
CDTIME_STATUS_STOPPED	2	CDTime is stopped (CDT hold).
CDTIME_STATUS_RUNNING	3	CDTime is counting.

COMMANDS

The CDTime commands are used to control the system Count Down Time (CDT) clock. These parameterized system commands are recognized by the CDTimeControl process. The four basic operations available are Set, Start, Stop and Cancel. The Command Descriptor Table includes the following system command FD’s:

	Command Parameters				
	Pre-defined		Variable		
Command FD Name	uint	uint	uint	int	int
1 Set_CDTime	Op_Code	Op_Opt	Time_Opt	Tsec	Tmsec
2 Start_CDTime	Op_Code	Op_Opt	Time_Opt	Tsec	Tmsec
3 Stop_CDTime	Op_Code	Op_Opt	Time_Opt	Tsec	Tmsec
4 Cancel_CDTime	Op_Code	Op_Opt	Time_Opt	Tsec	Tmsec

The CDTime system commands have six parameters, two are pre-defined. The first parameter is a pre-defined byte array used to clear the command parameters. The second parameter is a pre-defined integer used to specify the Op_Code. The next four command parameters are variable integers used to specify the operation option, time option, time seconds and time milliseconds. The state codes are defined in a system header file (see KConstants.h). The command parameters are defined as follows:

	Op_Code	Op_Opt	Time_Opt	Tsec	Tmsec
1.a	SET_CMD_OP	SET_CMD_OP	ELAPSED_TIME_OPT	secs	msecs
1.b	SET_CMD_OP	SET_CMD_OP	CDT_TIME_OPT	secs	msecs
1.c.*	SET_CMD_OP	SET_CMD_OP	GMT_TIME_OPT	secs	msecs
1.d.*	SET_CMD_OP	SET_CMD_OP	LOCAL_TIME_OPT	secs	msecs
2.a	START_CMD_OP	START_CMD_OP	ELAPSED_TIME_OPT	secs	msecs
2.b	START_CMD_OP	START_CMD_OP	GMT_TIME_OPT	secs	msecs
2.c	START_CMD_OP	START_CMD_OP	LOCAL_TIME_OPT	secs	msecs
3.a	STOP_CMD_OP	STOP_CMD_OP	ELAPSED_TIME_OPT	secs	msecs
3.b	STOP_CMD_OP	STOP_CMD_OP	CDT_TIME_OPT	secs	msecs
3.c	STOP_CMD_OP	STOP_CMD_OP	GMT_TIME_OPT	secs	msecs
3.d	STOP_CMD_OP	STOP_CMD_OP	LOCAL_TIME_OPT	secs	msecs
4.1.a	CANCEL_CMD_OP	SET_CMD_OP	ELAPSED_TIME_OPT	secs	msecs
4.2.a	CANCEL_CMD_OP	START_CMD_OP	N/A	N/A	N/A
4.3.a	CANCEL_CMD_OP	STOP_CMD_OP	ELAPSED_TIME_OPT	N/A	N/A
4.3.b	CANCEL_CMD_OP	STOP_CMD_OP	CDT_TIME_OPT	secs	msecs
4.3.c	CANCEL_CMD_OP	STOP_CMD_OP	GMT_TIME_OPT	secs	msecs
4.3.d	CANCEL_CMD_OP	STOP_CMD_OP	LOCAL_TIME_OPT	secs	msecs
4.4.a	CANCEL_CMD_OP	CANCEL_CMD_OP	N/A	N/A	N/A

State Definition	Value	Description
START_CMD_OP	1	Operation for a Start CDT Command
STOP_CMD_OP	2	Operation for a Stop CDT Command
SET_CMD_OP	5	Operation for a Set CDT Command
CANCEL_CMD_OP	6	Operation for a Cancel CDT Command
ELAPSED_TIME_OPT	101	Elapsed Time Value is specified
CDT_TIME_OPT	102	Count Down Time Value is specified
GMT_TIME_OPT	103	GMT(UTC) Time Value is specified
LOCAL_TIME_OPT	104	Local (system) Time is specified

CDTime Command Descriptions

1 SET_CMD_OP

The SET_CMD_OP commands are used to set the initial value of the CDT clock. A command to set the initial value of the CDT when it is already set is invalid and will generate an error. The CDT must be cancelled to change the initial CDT value. Multiple CDT clocks are not available.

1.a Set initial CDT to the elapsed time value specified (i.e. initial CDT is set to -Time_Val). Negative time values are invalid and converted to positive.

1.b Set initial CDT to the Count Down Time value specified. Positive and negative time values are valid.

1.c.* Set T0 to GMT time value specified. The CDT value will be calculated after the CDT is started so that T-Zero (CDT=0) occurs at the specified GMT. Past time values are invalid and will generate an error.

1.d.* Set T0 to local time value specified. The CDT value will be calculated after the CDT is started so that T-Zero (CDT=0) occurs at the specified local time. Past time values are invalid and will generate an error.

2 START_CMD_OP

The START_CMD_OP commands are used to start the CDT clock at current CDT value. A command to start the CDT clock when it is not set or is already started is invalid and will generate an error. A new start command will replace a pending start command. Multiple pending start commands are not available.

2.a Start CDT clock using current CDT value when elapsed time specified has expired. Negative time values are invalid and converted to positive.

2.b Start CDT clock using current CDT value when system clock reaches GMT specified. Past time values are invalid and will generate an error.

2.c Start CDT clock using current CDT value when system clock reaches local time specified. Past time values are invalid and will generate an error.

3 STOP_CMD_OP

The STOP_CMD_OP commands are used to stop the CDT clock at current CDT value. A command to stop the CDT clock when it is not set or is already stopped is invalid and will generate an error. Multiple pending stop commands are available.

3.a Stop CDT clock using current CDT value when elapsed time specified has expired. Negative time values are invalid and converted to positive.

3.b Stop CDT clock when CDT clock reaches CDT specified. Past time values are invalid and will generate an error.

3.c Stop CDT clock using current CDT value when system clock reaches GMT specified. Past time values are invalid and will generate an error.

3.d Stop CDT clock using current CDT value when system clock reaches local time specified. Past time values are invalid and will generate an error.

4 CANCEL_CMD_OP

The CANCEL_CMD_OP commands are used to cancel the CDT clock or any pending CDTime commands. A command to cancel the CDT clock when it is not set is invalid and will generate an error. A command to cancel a command that is not pending is invalid and will generate an error. The cancel commands use the 'Op_Opt' parameter to identify the command operation to be cancelled. The command to cancel CDT will reset the CDT to zero. A new cancel command will replace a pending cancel command. Multiple pending cancel commands are not available.

4.1.a Cancel the CDT clock and set the CDT value to zero when elapsed time specified has expired. Negative time values are invalid and converted to positive.

4.2.a Cancel a pending START_CMD_OP command to start the CDT clock.

4.3.a Cancel the next pending STOP_CMD_OP 3.a command.

4.3.b Cancel a pending STOP_CMD_OP 3.b command that matches the CDT specified.

4.3.c Cancel a pending STOP_CMD_OP 3.d command that matches the GMT specified.

4.3.d Cancel a pending STOP_CMD_OP 3.c command that matches the local time specified.

4.4.a Cancel a pending CANCEL_CMD_OP command to cancel the CDT clock.

RETURN VALUES

Returns 0 if no errors occur; 1 otherwise.

NAME

CctkClient - Graphical client for CCTK

SYNOPSIS

CctkClient

DESCRIPTION

CctkClient is a graphical client application that interfaces with the CCTK system. CctkClient allows the user to choose a project to interface with. CctkClient allows the user to start a project, stop a project, and view information about a project. CctkClient uses a display tree to present a list of valid displays to the user. The CctkClient display tree is configured in the project configuration file and dynamically at run-time.

When CctkClient is executed, it checks to see if the KPATH environment variable is defined. If the environment variable is defined, then CctkClient will check the directory listed by the KPATH environment variable to see if a valid project exists in the directory. If a valid project exists in the directory, it will be loaded.

OPTIONS

CctkClient does not support any options.

RETURN VALUES

CctkClient returns 0 if no errors occur during execution. A positive integer will be returned if an error occurs.

NAME

ChanMan - monitor channels

SYNOPSIS

ChanMan [kpath]

DESCRIPTION

ChanMan is an interactive display of the current status of the kernel channels. ChanMan must have a kpath to assist in accessing the channels. The kpath may be specified on the command line or by an environment variable named KPATH. ChanMan has a help screen which lists the options the user may enter to control the ChanMan display. Hit the "h" key to display the help screen.

RETURN VALUES

Upon normal termination, ChanMan returns zero. Otherwise, an error message is output and one is returned.

NAME

ChannelReader - Perform generic CLIENT processing.

SYNOPSIS

ChannelReader -c channel_name [-noout] [-d debug_flag] [-i packet_id] [-r register_fd_filename] [-f output_filename]

DESCRIPTION

ChannelReader acts as a generic CLIENT, providing channel read capablities. In normal usage, it reads packets over the specified channel, acquiring the time embedded in the packet and comparing it to the current time to determine latency. If the "-r" option is specified, ChannelReader registers for the FD names contained within the register_fd_filename. ChannelReader will output the packets which are received to stdout, unless the capability is explicitly inhibited during the startup of ChannelReader (-noout). It also supports a DEBUG mode (-d TRUE/FALSE). ChannelReader may be used as a CLIENT without using a testing SERVER process. If the packets are larger than the standard kernel packet_hdr (see KTypes.h), ChannelReader expects to find timetag information within the packet header per the standard KERNEL header definition. If the timetag is not present in the appropriate location, strange latency values will be recorded. In order to terminate ChannelReader, a "kill -15" (SIG_KTERM) may be used; ChannelReader will terminate normally when it detects that the SERVER has closed the channel, after at least one packet has been read.

OPTIONS

-c channel_name

channel_name is the name of the channel which is providing the inputs to ChannelReader. This input is mandatory. channel_name may be a temporary channel; this is done by specifying "TEMP" as the channel name. In this case, packet_id must be entered. ChannelReader, in its standard operating mode, outputs packet information to stdout. This must be redirected to a file.

-noout

The noout option inhibits the output to stdout.

-d debug_flag

debug_flag indicates whether the debug is to be active during execution. This value may be TRUE or FALSE; default = FALSE.

-i packet_id

packet_id specifies the ID of the packet which is to be used on the STDDIST channel to specify the name of the temporary channel to the SERVER process. This is required if a temporary channel is specified. ChannelReader (and ChannelWriter) assume that the format of the STDDIST packet is the standard KERNEL packet header, followed by the null-terminated string containing the name of the temporary channel.

-r register_fd_filename

This option is used to specify a file which contains a list of the FD's for which ChannelReader is to register. The file should contain one FD name per line, no extra text on each line. The file may contain comments, using the script file comment character (#) as the first character in each line. The comments may be located anywhere within the file.

-f output_filename

This option allows the user to specify that the output is to go to output_filename. If -noout is specified, no information will be output to the file. If this option is not specified, AND if -noout is not specified, output will go to stdout.

SEE ALSO

ChannelWriter

NAME

ChannelWriter - Write packets onto a KERNEL channel per commands specified in a script file, using packets contained within packet files.

SYNOPSIS

ChannelWriter -s script_filename -c channel_name [-i packet_id] [-d g_debug]

DESCRIPTION

ChannelWriter, in union with its cohort, ChannelReader, support component and higher levels of testing by providing a script- driven process which enables the tester to transmit channel packets at various rates. NOTE: ChannelWriter does not require that ChannelReader be running as a CLIENT on the other side of a set of processes; neither does ChannelReader require one (or more) ChannelWriters.

The tester is able to select between packet transmission at a desired transmit rate, or allowing ChannelWriter/ChannelReader to determine and report on the maximum sustainable rate within the current environment. When ChannelWriter execution is complete, ChannelWriter will terminate, and ChannelReader will report on the average, highest, and lowest latency which was achieved.

ChannelWriter requires the input of the following:

OPTIONS

-s script_filename

The name of the script which contains the commands necessry for the execution of this instance of ChannelWriter.

-c channel_name

The name of an authorized channel. If the channel is a temporary channel, specify channel name TEMP. In this case, the packet ID which the server registers for to obtain the temporary channel name must also be specified. This component assumes that the packet which will be received has the following format: standard KERNEL packet header NULL-terminated string containing the name of the temporary channel.

-d debug_flag

debug flag, Default = FALSE, value must be [TRUE | FALSE]

SCRIPT LANGUAGE

The following commands are available in the ChannelWriter script files. The commands are case-sensitive. The format of the commands within the script file is:

```
# sequence_num command
```

where sequence_num is an integer. sequence_num will be used to determine the next command to be executed after a GO_TO command. A pound sign (#) is used as a comment character; empty lines are also legit.

START_SCRIPT

All scripts are required to start with this statement. This allows ChannelWriter to detect the difference between the start of the ChannelWriter script, and a simple UNIX comment line. A "END_SCRIPT" is not required (nor does one exist): ChannelWriter assumes that EOF on the script file, or a line which does not begin with a pound sign, indicates the end of the script.

OPEN filename

This command opens a file which contains channel packets. Up to 10 files may be open at any time.

CLOSE filename

This command closes a file which contains channel packets.

SUPERPACKET n

This command flags ChannelWriter that a superpacket is to be generated, with "n" subpackets. The rules for its use are:

1. The READs which fill the superpacket(s) must follow immediately after the SUPERPACKET command. Any other command turns off the SUPERPACKET flag.
2. ChannelWriter will perform the READs, sticking "n" packets at a time into the superpackets.
3. The final READ does not need to ensure that the superpacket contains "n" packets. ChannelWriter keeps track of the number of subpackets automatically.
4. A superpacket is treated as a single packet, so a FLUSH will clean out the entire superpacket, and a TRANSMIT will transmit the entire superpacket.

READ n filename

This command causes n packets to be read into the packet buffer. The packet buffer contains up to 20 packets. The packets will be stored into the buffer sequentially as they are read; they will be stored after any packets which already exist in the buffer. These packets should have the standard KERNEL header structure. ChannelWriter assumes the standard structure, which includes locations for a timestamp, and writes the current time information into that location within the packet during execution of the TRANSMIT command. This is the only modification of the packet's contents performed by ChannelWriter.

TRANSMIT n-m t r

This command transmits packets n through m (first packet in the buffer is number 0) on the channel. The duration of the transmission = t seconds (MUST be > 0), and packets are to be transmitted r times per second (0 indicates that ChannelWriter is to perform at the maximum sustained pace over the specified duration.) Therefore, if you've specified packets 0-4 at a rate of 5, the aggregate rate will be 25 packets per second. ChannelWriter currently uses a "rough" transmission method: the packets are burst onto the network at the maximum transmission rate; the transmitter then waits until the remainder of the current clock interval (1 second) has elapsed.

FLUSH n

This command causes packet # n to be flushed from the buffer. The first packet in the buffer is number 0. All packets subsequent to packet # n in the buffer will be moved to fill up the blank space.

SLEEP x

This command sleeps for duration x, where x is a integer value > 0 which indicates the number of MICROSECONDS which are to be slept.

GO_TO seqnum

This command performs a direct "branch" to the sequence number specified by seqnum. The first instance of seqnum in the script file will be found; execution will resume starting with the command associated with seqnum.

SCRIPT startup_string

This command allows you to start up anything else that you desire. The rest of the line, up to the first instance of the comment character, is used to initiate another process. If the last non-blank character is an ampersand, the activity will run in the background of the current process. Otherwise, if the last non-blank character is not an ampersand, the specified activity will execute, and execution of the current script will pause until the execution of startup_string is complete (think of it as a subroutine).

SAMPLE SCRIPT FILE

```
(((((((((((((((((((((((((((((((((((((((())))))))))))))))))))))))))))))))
))))))) # r_sccs= " @(#)ChannelWriter.c 29.1 96/02/27 13:22:35:"

# Start the ChannelReader, DataProc, and then the ChannelWriter

ChannelReader -c DataProcChannel > ChannelReader.testout &
DataProc DataProcChannel &
ChannelWriter -c DataProcInChannel -s $0

# START_SCRIPT
## This script opens a file and reads in 5 packets.
## It transmits the last 3 packets for 2 seconds; the aggregate rate
## is 3000 Hz
## It transmits all 5 packets for 1 second at an aggregate rate of 5
```

```
        hz ## It flushes the last 3 packets and reads in 3 more
        ## It then goes to the transmit statement
        ## This loop will continue until an EOF is hit in the input file,
        ## at which point ChannelWriter will terminate.
        #1 OPEN data_proc_script_1
        #1 READ 5 data_proc_packets_1
        #2 TRANSMIT 2-4 2 1000
        #2 TRANSMIT 0-4 1 1
        #2 FLUSH 4
        #2 FLUSH 3
        #2 FLUSH 2
        #2 READ 3 data_proc_packets_1
        #2 GO_TO 2
        od -x ChannelReader.testout | less
        ((((((((((((((((((((((((((((((((((((((((()))))))))))))))))))))))))))))))))
 )))))))
```

SAMPLE SCRIPT FILE

```
        ((((((((((((((((((((((((((((((((((((((((()))))))))))))))))))))))))))))))))
        ))))))) # r_sccs= " @(#)ChannelWriter.c 29.1 96/02/27 13:22:35:"

        # Start the ChannelReader, DataProc, and then the ChannelWriter

        ChannelReader -c DataProcChannel > ChannelReader.testout &
        DataProc DataProcChannel &
        ChannelWriter -c DataProcInChannel -s $0

        # START_SCRIPT
        ## This script opens a file and reads in 5 packets into 2 superpackets.
        ## It transmits the 1st superpacket for 2 seconds
        ## It transmits both superpackets for 1 second
        ## It flushes the superpackets and reads in 2 regular packets
        ## It then goes to the transmit statement
        ## This loop will continue until an EOF is hit in the input file,
        ## at which point ChannelWriter will terminate.
        #1 OPEN data_proc_script_1
        ## Create 2 superpackets, max size = 3 subpackets, so the 1st has
        ## 3 subpackets, the 2nd superpacket has 2 subpackets
        #1 SUPERPACKET 3
        #1 READ 5 data_proc_packets_1
        #2 TRANSMIT 0-0 2 1000
        #2 TRANSMIT 0-1 1 1
        #2 FLUSH 1
        #2 FLUSH 0
        #2 READ 2 data_proc_packets_1
        #2 GO_TO 2
        od -x ChannelReader.testout | less
        ((((((((((((((((((((((((((((((((((((((((()))))))))))))))))))))))))))))))))
 )))))))
```

SEE ALSO

ChannelReader

NAME

ClearSystemStateOnBoot - Script for use in system start scripts to automatically reset the system state.

SYNOPSIS

ClearSystemStateOnBoot <path to CCTK direcotry> <project directory> [project directory] [...]

DESCRIPTION

ClearSystemStateOnBoot is a simple shell script which calls SetSystemState for a series of projects. If the state of any of those projects is not DOWN, ClearSystemStateOnBoot will change the project state to down. ClearSystemStateOnBoot is provided as a simple way to ensure the state of a project is either DOWN or FORCED_DOWN during system boot. This is necessary if the system crashed (i.e. power outage) while a CCTK project was in the UP state.

OPTIONS

The first option is the path to the CCTK directory (aka CCT_HOME). This is necessary so that ClearSystemStateOnBoot can find the CCTK binaries. The remaining options are the paths to the project directories to check. Globbing is your friend if you keep multiple projects in a single directory.

SEE ALSO

SetSystemState, SystemState

NAME

DataProc - Kernel Data Processor

SYNOPSIS

DataProc -i input_channel_name [-o output_channel_name] [-k kpath] -t table_name [[-t table_name]...] [[-l shlib] ...]

OPTIONS

-i input_chan_name

is the name of the channel where DataProc receives raw data packets for processing.

-l shlib

is the name of any additional shared libraries to be used by DataProc.
More than one library name could be specified.
Each would contain the user defined processing functions that can be utilized by this DataProc.

-o output_chan_name

is the name of the channel where processed data packets or exception messages generated by DataProc are sent. This is typically used when an application wishes to receive time-corrleated measurements sent by an interface as superpackets. See the Administrator's guide for a detailed explanation.

-n

tells DataProc to NOT distribute processed data packets and exception messages. Without this flag, DataProc will distribute processed data packets and exception messages to any process which registers for them. If this flag is specified, DataProc will not perform distribution. NOTE: If this flag is present, data WILL NOT be archived via the standard methods.

-k kpath

specifies the value to be used for the KPATH environment variable. Default KPATH taken from the shell environment.

-t table_name

is name of a realtime table used by DataProc. More than one table could be used. They would contain the descriptors for all measurements that can be processed by this DataProc. Requires at least one realtime table; typically, that is the "mdt" Maximum number of tables allowed is MAX_NUM_DATAPROC_TABLES.

DESCRIPTION

The DataProc is a Kernel process that is responsible for data conversion, compression(filtering), valid range checking, max change checking, and exceptions checking for measurements.

Input to DataProc are in the form of linked_data packets which contain the raw values for the measurements. These packets can come in singly or grouped together in a superpacket. This is the the structure for a superpacket:

```
struct super_packet {
        packet_hdr_t    packet_header;
        linked_data_t   raw_data_packet[...];
}
```

Raw measurement input is processed according to the DATA_PROCESSING_ OPTIONS defined in that measurement's descriptor and according to its data type. The data types currently supported by DataProc are: ANALOG, INTEGER, DISCRETE and BYTE_ARRAY.

Processing of the linked data produces processed_data packets. Exception_messages can also be generated as the result of processing. For each exception packet, there must also be an accompanying processed_data packet if the data is able to be processed. All processed_data packets and exception_messages generated for measurements that came in as a super packet will be group together in an output super packet for distribution. If processing resulted in only one processed packet, this packet will be transmitted without being packaged in a superpacket.

In the situation when the generated processed packets can not fit into one output super packet, they will be split into several super packets.

Unrecognizable or invalid raw data are not processed. An invalid data exception message is generated for each valid->invalid transition.

DATA_PROCESSING_OPTIONS

For each measurement, various processing options can be enabled or disabled by using the configuration database to change the measurement descriptor. Parameters associated with these options are also in the descriptor. This is how DataProc operates when the option is enabled:

DATA_COMPRESSION - DataProc checks for whether the measurement had changed by a significant amount since a last sample. The significant change criteria can be based on the raw value, the processed value or a percentage of the processed value. If significant change criteria is not met, output of processed data packet is suppressed.

To avoid creeping significant changes that will not be detectable, the last sample used is the last one that had a significant change.

STALE_DATA_CHECK - DataProc checks whether the raw value had changed. If it had not changed after a certain number of cycles, the data is considered stale. It shall remain stale until the raw value changes. The number of cycles or the stale sample threshold is defined by the user.

DATA_CONVERSION - applicable only to analog measurements. DataProc will convert the raw data value into a processed data value based on the algorithm specified by the measurement's descriptor.

If the algorithm is POLYNOMIAL_CONVERT, the raw data is treated as an integer raw count value to be used with the polynomial equation defined for that measurement to calculate the processsed value.

If not enabled, or if no conversion algorithm is specified, DataProc uses the raw data value as the processed data value. The raw value is assumed to be in IEEE double precision floating point format.

If error is encountered when attempting the conversion, the CNVRT_ERR_BIT_MASK bit shall be set for the measurement's status and its value is invalid.

VALID_RANGE_CHECK - DataProc verifies that the data sample does not exceed the defined upper or lower range values. The criteria for valid range can be based on the raw value or the processed value. If one of these range values is violated, the INVALID_RANGE_ERR status for the measurement is set and other exception checks will not be performed. This option is not applicable to DISCRETE or BYTE_ARRAY measurements.

MAX_CHANGE_CHECK - DataProc verifies that the current data sample and the last sample's does not differ by more than a defined delta value. The criteria for this delta value can be based on the raw value, the processed value or a percentage of the processed value. If this condition is violated, the MAX_CHANGE_ERR status bit for the measurement is set. This option is not applicable to DISCRETE or BYTE_ARRAY measurements.

LIMIT_CONDITION_CHECK - the processed value is used to verify that a certain condition had occurred. Each condition is composed of a pair of limit logic definitions (limit, logic_condition). Normally, only one definition is used to specify the limit condition. When the limit condition needs to be a range, the two definitions can be combined via AND/OR relationship to specify it. The legal logic_conditions are: = , != , > , >= , < , <= .

If the processed value satisfies the condition then the LIMIT_CONDITION_ERR status bit for this condition is set for the measurement's status.

There could be up to 8 such conditions per sample. DISCRETE data type can only have 1 limit condition with only 1 limit logic definition.

EXAMPLE

This is used to detect when measurement X's value is greater than 100: Condition 1: [limit_1 = 100, logic_cond_1: '=']

This is used to detect when measurement X's value is between 20 and 50 inclusively: Condition 1: [limit_1 = 20, logic_cond_1: '>='] AND [limit_2 = 50, logic_cond_2: '<=']

EXCEPTION MESSAGES

An exception packet is generated when there is a change in these status bits of the processed value: INVALID_RANGE, MAX_CHANGE_ERR, LIMIT_CONDITION_ERR and DATA_STALE.

The exception packet contains the current and previous data statuses. It is up to the end-user to compare these statuses to determine the cause(s) of the exception or to detect if an exception condition has returned to normal.

An exception packet is always accompanied by a processed data packet if the data was able to be processed.

DIAGNOSTICS

DataProc will terminate if input or output channel was not specified or if initialization was unsuccessful.

NAME

Distributor - read packets from one channel and write them to all the channels that have registered to receive them and to any channels specifically identified in the packet itself.

SYNOPSIS

Distributor -i input_channel_name [-k kpath]

DESCRIPTION

Distributor reads packets from the channel specified by input_channel_name and outputs the packets to all channels that have registered to receive that packet via the function CDistCntl(). The packets are also written to any channels that are specified in the distribution vector contained in the packet header.

OPTIONS

-i input_channel_name

the channel from which packets will be read.

-k kpath

The Distributor process must have a kpath to assist in accessing the channels. The kpath may be specified on the command line or by an environment variable named KPATH.

NOTES

If an input packet is a super packet which contains more than one subpacket, the super packet is separated into the subpackets which are distributed individually.

RETURN VALUE

Upon normal termination, Distributor returns zero. Otherwise, an error message is output and one is returned.

SEE ALSO

CDistWrite(), CDistCntl()

NAME

HistRetrieve - provides historical retrievals for CCTK

SYNOPSIS

HistRetrieve [-i|v|q] [-D] [-s[s]] [-p path] [-o output_file] [-b start_time] [-e end_time] [-f filebase] parameter_file [parameter_file ...]

DESCRIPTION

This process provides for the Historical retrieval of data archived by the ARS process. HistRetrieve works similarly to NrtRetrieve. Please read the NrtRetrieve manpage for more details. Only the difference between it and HistRetrieve will be described here.

HistRetrieve does not support the -t (timeout) option that NrtRetrieve does. If HistRetrieve is passed this option, it will produce an error.

HistRetrieve supports two additional, although optional, arguments. The first, -p, indicates the path where HistRetrieve should look for the historical archive files. If it is not specified, it defaults to the current directory. The second, -f, indicates the base filename of the archive files. If not specified, this defaults to TAM.

The -s option is also different from NrtRetrieve. -s cause HistRetrieve to print out the statistics associated with a historical retrieval. It displays the FD name, the earliest recorded measurement, the latest recorded measurement, and whether or not compression was enabled. If a second -s option is given, it will additional print any information known about the Fds in the archive.

RETURN VALUES

HistRetrieve supports the same error codes as NrtRetrieve. Please see it for a description.

SEE ALSO

NrtRetrieve

NAME

MeasMon - Generic Measurement Monitor for CCTK.

SYNOPSIS

MeasMon [-u update_rate] [file to load]

DESCRIPTION

MeasMon is short for Measurement Monitor. It is a generic interface for viewing measurements within CCTK. MeasMon will allow the user to view any measurement in a neat tabular format.

MeasMon provides controls for the following:

Save/Load Files
Add/Remove FD's from the tabular list
Freeze the display (stops updates)
Toggle between GMT/Local time

When saving and loading files, all valid information about the GUI is saved. This not only includes the list of Fds being displayed, but also the fields being displayed, the width of the fields, the state of the time, and the format of the time.

Note:

At this time, the user cannot change the format of the time field or the fields being displayed. These can only be changed by hand modifying the saved file.

OPTIONS

-u update_rate

The update rate can be specified on the command line. The update rate states how long between each display update cycle. The default is once per second.

NAME

Message - notice processor

SYNOPSIS

Message

DESCRIPTION

Message is a kernel process designed to provide centralized notice generation and routing facilities. Message is to be utilized by all kernel processes, and is available to any utility processes via the SendCond() interface function.

SendCond() will generate and transmit via communications channel to the Message process, a packet that can contain one or several condition_notices (or a superpacket).

Each type of condition_notice has a corresponding notice descriptor definition in the notice table in the kernel real-time table database.

Message will analyze the notice descriptor for the current condition_notice and perform processing appropriate for the current notice descriptor definition.

Message's primary task is to generate and issue kernel system notices as directed by the condition_notice and its notice descriptor. System notices may be routed to distribution, to the kernel system log, or both. Suppression of system notice generation is also allowable.

Message will generate a system notice using a formatted text string contained in the notice descriptor.

System notice for condition_notices that are from the same input superpacket will also be kept together in the same system notice superpacket for distribution.

The KPATH environment must be established prior to the execution of this process.

ARGUMENTS

None required.

DIAGNOSTICS

Message will terminate upon failure to acquire channel resources or notice table memory access.

SEE ALSO

Channel, SendCond

NAME

MonCon - Generic monitor and control application

SYNOPSIS

MonCon [-f] <drawing filename> [-i] [-u <data update rate>] [-r <display refresh rate>]

DESCRIPTION

The MonCon application provides a generic mechanism for binding graphical monitor and control functionality to the measurements and command database.

OPTIONS

-f drawing filename

Drawing filename to load. (the -f switch is optional)

-i input filename

Setup data filename associated with the drawing.

Note:
If a data filename is not specified then the data filename is assumed to be the drawing filename where any ".???" extension is replaced by a ".dat".

-u data update rate
Data update rate in milliseconds (rate which data is polled). (Default: 1000 milliseconds or once per second)

-r display refresh rate
Refresh rate in milliseconds (rate which display is redrawn). (Default 100 milliseconds or ten times per second)

-geometry geometry
Specifies the initial size and location of the drawing.

See moncon_example.dat setup file located in the examples directory for details regarding application setup.

See libmoncon.a description for details regarding specific display and application functionality.

RETURN VALUES

If an error occurs during initialization or termination then MonCon exits with a value of -1. Otherwise, upon successful termination a value of 0 is returned.

NAME

MulticastServer - Generates packets based upon the MulticastProtocol developed for CCTK.

SYNOPSIS

MulticastServer [-a] [-m ADDRESS|HOST] [-p NUM|SERVICE] [-t NUM] [-l] [-r FLOAT] [-i NAME] [-P NAME] [-s NAME] [-k PATH]

DESCRIPTION

MulticastServer will generate multicast packets for clients. The protocol used is described in the MulticastProtocol.h header file in the CCTK include directory. MulticastServer provides two-way communications with the clients where the clients can register for measurements to receive, interact with the server, and receive data. The multicast-address and the multicast-port which the packets are sent out are specified on the command line.

The preferred way to specify the address is to place an entry in the hosts() file that references the correct IP address. MulticastServer will use gethostbyname() to lookup the address. The preferred way to specify the port is to place an entry in the services() file that references the correct port. MulticastServer will use getservbyname() to lookup the port. Raw addresses and ports can be used as well as the lookup methods.

A time-to-live parameter can also be specified on the command line. This defines for how long this packet is allowed to live on the internet, see multicast documentation for a further description of its meaning. This value should ALWAYS be set to the default of 1 unless you are distributing packets via multicast to clients that are not on your local network.

MulticastServer will receive measurement information and commands from CCTK. It will also receive requests via multicast from the clients. These requests will cause MulticastServer to register for CCTK measurements. Each time one of these measurements is received, it will packetized and distributed to the clients via multicast. For proper command routing, the port-fd must be specified. MulticastServer will place its input channel distribution vector in the specified port descriptor so that command routing occurs correctly. If no port descriptor is specified, MulticastServer will not be commandable by the standard CCTK methods. The CCTK KPATH can also be specified using the KPATH option.

Packet generation can be activated/deactivated in one of two ways. First, MulticastServer will respond to CCTK commands that perform the task. Second, MulticastServer will respond to the UNIX signals TSTP and CONT. These signals will cause MulticastServer to deactivate and activate packet generation respectively. The UNIX signals are mainly useful in testing.

MulticastServer provides several debugging utilities. First, MulticastServer can ALWAYS be forced to output its current set of status statistics by sending it the SIGHUP signal. Second, if debug is compiled in, a debug level can be set upon startup. The debug level is from 1 to 5 with level 1 providing a minimum amount of data while level 5 produces large amounts of data. If debugging is specified and debugging is not compiled in, a warning message will be generated.

MulticastServer can be shutdown by sending it a TERM or an INT signal.

OPTIONS

-a

This flag determines the state that MulticastServer initializes in. If the flag is set, MulticastServer will come up and immediately start sending packets. If the flag is not set, MulticastServer will need to receive a command before it starts sending packets. This is an optional parameter. Multiple occurrences are ignored.

-i NAME

NAME will specify the input channel name to use. The input channel will be used by MulticastServer to receive both registered measurement data and commands. It is an optional parameter. It may only be specified once. If it is not specified, the NEXT_AVAIL_CHAN will be used.

-k PATH
PATH will be used as the CCTK KPATH for this process. The path must contain a valid, running CCTK system. It is an optional requirement. If KPATH is specified, the "KPATH" environment variable will be set to PATH. If KPATH is not specified, KPATH will be retrieved from the "KPATH" environment variable. If the environment variable is not present and KPATH is not set on the command line, an error will be generated.

-m ADDRESS|HOST
ADDRESS or HOST will establish the multicast address over which the data packets will be transmitted. It is an optional parameter. It may only be specified once. It must either be a valid address of the format X.X.X.X or a valid hostname that can be retrieved via gethostbyname() system call. It must be between the values of 224.0.0.0 and 239.255.255.255. If not specified, the default in the MulticastProtocol.h file will be used.

-p NUM|SERVICE
NUM or SERVICE will establish the multicast port over which the data packets will be transmitted. It is an optional parameter. It may only be specified once. It must either be a valid port number or the name of a service which can be retrieved by the getservbyname() system call. It must be greater than 0 and less then the maximum port allowable by the system. If not specified, the default in the MulticastProtocol.h file will be used.

-P NAME
NAME will specify a port FD to use. The input channel of MulticastServer will be registered in the specified port-fd so that commands are routed correctly. It is an optional parameter. It may only be specified once. If no port is specified, then MulticastServer will not be commandable via the normal CCTK methods.

-r FLOAT
FLOAT will establish the packet rate at which MulticastServer transmits packets. It is in Hz. It is an optional parameter. It may only be specified once. It must be greater than 0.0 but less than 100.0, creating a maximum transmit rate of 100 Hz. If packet-rate is not specified, it will default to 1.0. MulticastServer will transmit faster than this rate if many measurement updates are received. MulticastServer only guarantees that it will at least transmit at this rate.

-s NAME
NAME will specify the status channel name to use. The status channel will be the channel that status packets are sent to. It is an optional parameter. It may only be specified once. If it is not specified, STDSTATUS will be used.

-t NUM
NUM will establish the time-to-live value placed in the header of each multicast packet. It is an optional parameter. It may only be specified once. It must be greater than or equal to 1 and less than 255. If time-to-live is not specified, it will default to 1.

-l
Enables the IP multicast loopback. This must be enabled if clients will be running on the same machine as the server. Enabling this will increase the load on the server.

-d level
If this option is specified, MulticastServer will display debugging information. MulticastServer will display more information as the debug level increases. Valid debug levels are from 1 to 5. Please note that a debug level above 2 will generate enormous amounts of data. It is possible that the debug code has been removed from the application. If this is the case, then a warning message will be displayed and this option will be ignored. This option should mostly be used for testing.

-h
This will produce a brief help listing on the use of MulticastServer.

DIAGNOSTICS

MulticastServer will exit with a 0 return code upon a successful termination. If any errors occur during initialization due to incorrect arguments, it will exit with a 1. If any errors occur during initialization due to problems parsing the mpd file, it will exit with a 2. If any errors occur during initialization due to problems setting up the system, it will exit with a 3. If any unrecoverable errors occur during operation, it will exit with a 4.

Also, after communications have been established with CCTK, MulticastServer will send System Messages if any critical or cautionary errors occur.

SEE ALSO

MulticastServer.h

NAME

NrtRetrieve - provides near real-time retrievals for CCTK

SYNOPSIS

NrtRetrieve [-i|v|q|] [-D] [-k kpath] [-o output_file] [-b start_time] [-e end_time] [-t timeout] parameter_file [parameter_file ...]

DESCRIPTION

NrtRetrieve() is responsible for performing a near real-time retrieval for the CCTK system based upon a given parameter file. NrtRetrieve will process each parameter_file listed on the command line in turn and perform a retrieval based upon its contents.

Each parameter file must be of the correct format. The format for the parameter files are described in the retrieval parameter file definition document. Please see that document for more details. If the -o option is specified, it overrides any filenames specified in the parameter file. If a single parameter file is specified and the -o option is specified, the output will be in the filename specified by -o. If multiple parameter files are specified and the -o option is specified, the output will be in the filename specified by -o with a sequence number appended to the name. For example, if three parameter files were specified, the output files would be:

```
output_file.0
output_file.1
output_file.2
```

If the -b and/or -e options are used to specify a start/stop time, then they will override any times specified in the parameter file. Simple checks will be done to ensure that the start time occurs before the stop time. An error will be returned if this is not the case. If no start/stop times are specified, ARS will try to use the entire length of the existing archive.

The -i, -v, and -q options are mutually exclusive. They determine the amount of feedback provided by NrtRetrieve to the caller. Only one of these options may be specified on the command line. An error will be generated if more than one is set.

NrtRetrieve will accept several signals. Whenever NrtRetrieve receives the CONT signal, it will attempt to continue with the retrieval (unpause). Whenever NrtRetrieve receives the TSTP signal, it will attempt to pause the retrieval. Finally, whenever NrtRetrieve receives the INTR signal, it will attempt to abort the retrieval.

OPTIONS

-i

is used to indicate that the application is being called interactively on the command line. In this case, status on the retrievals will be printed to the screen in a user friendly format. Displaying both % complete and number of samples retrieved.

-v

is used to indicate that the application is being called by another application which desires feedback, but does not need the user friendly, screen formatted, output produced by the -i option. In the -v mode, the following output will be generated to standard out:

total:<number> – The number of retrievals for this session.
param:<pid>:<file_name> – This line will be sent each time processing on a new parameter file is started.
output:<file_name> – The name of the output file associated with the current parameter file.
state:<current_state>:<description of state> – This line will be sent each time the "state" of the retrieval changes. Valid states are init, pause, exec, complete, fail, abort.
percent:<percent complete> – This line will be sent periodically to indicate the percentage of the retrieval that is complete.
samples:<samples found> – This line will be sent periodically to indicate the number of samples found.

pid is the pid of the NrtProcess performing the retrieval. This pid can be used to send signals to the NrtRetrieve process to pause/continue/abort the retrieval.

-q

is used to quiet all output. No status messages will be displayed to standard out. This option will find use in batch retrievals. Error messages and codes will still be produced.

-T

is used to specify a timeout. The timeout is the maximum time to wait for a free archival slot before giving up.

-D

indicates that the input parameter file to NrtRetrieve is temporary and should be deleted once the Retrieval is finished. This is a kludge to allow the tmp directory to be cleaned up.

RETURN VALUES

Upon successful completion of the retrieval(s), an exit code of 0 will be returned. If NrtRetrieve fails, one of the following error codes will be returned to indicate the reason.

1. Invalid command line arguments.
2. Internal error occurred before attempting to initiate the retrieval.
3. This could be an indicator of several items, unable to read parameter file, unable to obtain necessary resources, unable to communicate with the test in the specified KPATH.
4. Unable to initiate retrieval within the given timeout period. This typically indicates that the ARS subsystem was too busy to handle the request.
5. Unable to start retrieval. This typically indicates that the ARS subsystem was "unable" to start the retrieval request.
6. Error during retrieval. This indicates that the retrieval was started successfully, but an error occurred while the retrieval was being processed.
7. Retrieval aborted. This indicates the retrieval was aborted by the user.

SEE ALSO

TamArs, HistRetrieve

NAME

PeerReceiver - Receives CCTK data across a network

SYNOPSIS

```
PeerReceiver [-d LEVEL] [-i NAME] [-o NAME] [-S NAME] [-k PATH] [-s] [-p] [-f]
               [-P NAME | ([-m MODE] [-l ADRESSS] [-L PORT] [-r ADDRESS] [-R PORT])]

    or

PeerReceiver -h
```

DESCRIPTION

PeerReceiver receives CCTK data packets across the network using either TCP or UDP sockets. This application is typically used with PeerSender, but does not have to be. All types of CCTK packets can be received from processed data packets to linked data packets to packet decom packets. In its most basic form, PeerReceiver only examines the packet header of the packet and places the packet on the output channel so any and all future packets will be supported as well. Special packets allow PeerReceiver to associate named measurements from the sender system to named measurements on the local system. In addition, PeerReceiver also has the option to translate processed data packets to linked data packets to allow processed data packets from one CCTK system to be processed by another. The protocol used by PeerReceiver is open and thus anyone can read data sent by the process. The protocol is documented in the PeerProtocol.h header file. Also see the CCTK Administrators Guide for more information on configuring and using the PeerSender and PeerReceiver processes.

FD AND DATA TRANSLATION

PeerReceiver has several command line flags that controls how it translates data that it receives. PeerReceiver receives CCTK packets via the Ethernet connection. Each packet contains a sid that is a unique identifier for the FD for the project on the sender host. That sid may or may not be valid for the project running on the local, receiving host. Three sid translation methods are available:

1) SID lookup translation
2) Direct FD translation
3) No translation

The first method, which is active if a port descriptor is specified on the command line, causes a lookup to occur. The port descriptor defines a FD name mapping between FD names on the sender system and FD names on the local system. Therefore, when a FD is received, a lookup is performed. If a match is found, the local sid found there is used for the translation. This is the default translation if a port descriptor is defined. This method can be turned off by specifying the -s flag.

The second method, which is active by default, causes a direct FD translation to occur. For each FD received from the sender system, an FdToSid() is performed on the local system. The value returned by the FdToSid() call is used as the local sid for that FD. Therefore, anytime a packet is receive with the sender sid, the local sid will be substituted. The direct FD method will only be used if a translation using the first method fails. This is the default translation if no port descriptor is defined. This method can be turned off by specifying the -f flag.

The third method, which is active if the -s and -f flags are set on the command line, causes no translation to occur. In this case, it is assumed that the sids for FDs on the sender system match the sids for the FDs on the local system. This is only true if the configuration databases on the two systems are identical.

MODES OF OPERATION

PeerReceivere can receive data in one of five modes of operation. The modes of operation are:

- Registration
- Point-to-Point
- Connectionless

- Broadcast
- Multicast

Each mode of operation indicates how the socket should be configured to receive the data. The only special action taken by PeerReceiver based upon the mode is that in registration mode, a registration request will be made for each FD listed in the port table.

CONFIGURATION

PeerReceiver can be configured through a port descriptor or using command line options. If a port descriptor is specified (with the -P option), then it is not permissible to specify address information on the command (with the -l, -L, -r, or -R options).

The port descriptor is populated via the CCTK configuration database. Please see the peer-to-peer documentation for more information on configuring the PeerReceiver via the configuration database. The command line options are described below.

OPTIONS

-f
This option indicates that direct FD translation should not be performed. See the section above on FD and data translation for more information.

-i NAME
NAME will specify the input channel name to use. The input channel will be used by PeerReceiver to receive commands. It is an optional parameter. It may only be specified once. If it is not specified, it will default to NEXT_AVAIL_CHAN.

-k PATH
PATH will be used as the CCTK KPATH for this process. The specified path must contain a valid, running CCTK system. It is an optional requirement. If kpath is specified, the "KPATH" environment variable will be set to PATH. If kpath is not specified, kpath will be retrieved from the "KPATH" environment variable. If the environment variable is not present and kpath is not set on the command line, an error will be generated.

-l ADDRESS
Specifies the local address to use for the socket. ADDRESS should be of the standard form x.x.x.x or a hostname resolvable into an address using gethostbyname(). The local address is the address used to bind to on the sender host. This is optional for all modes. If not specified, any address will be used. It may only be specified once.
-L PORT
Specifies the local port to use for the socket. PORT should be a standard number or a service resolvable into a port using getservbyname(). This option is required for connectionless, broadcast, and multicast, optional for the other modes. If not specified, any available port on the system will be used. It may only be specified once.
-m MODE
MODE indicates the mode of operation for PeerSender. Valid modes of operation include: registration, point-to-point, connectionless, broadcast, and multicast. Each mode of operation requires different port and address information. The chart provided in the description above indicates which arguments are required for which mode. A further description of the modes of operation are provided above. This option cannot be specified more than once.

-o NAME
NAME will specify the output channel name to use. The output channel will be used by PeerReceiver to send data. It is a required parameter for all modes. It may only be specified once.

-p
This option indicates that packet translation should be performed. See the section above on FD and data translation for more information.

-P NAME

NAME will specify a port FD to use. The input channel of PeerSender will be registered in the specified port fd so that commands are routed correctly. It is an optional parameter. It may only be specified once. If no port is specified, then PeerSender will not be commandable via the CCTK system commands. If a port fd is specified, address and port information will be retrieved from the port descriptor and an error will be generated if address and port information is specified on the command line.

-r ADDRESS
Specifies the remote address to use for the socket. ADDRESS should be of the standard form x.x.x.x or a hostname resolvable into an address using gethostbyname(). This option is optional for connectionless, broadcast, and multicast. It is required by point-to-point and registration. It may not be specified more than once.
-R PORT
Specifies the remote port to use for the socket. PORT should be a standard number or a service resolvable into a port using getservbyname(). This option is optional for connectionless, broadcast, and multicast. It is required by point-to-point and registration. It may not be specified more than once.

-s
This option indicates that sid lookup translation should be performed. See the section above on FD and data translation for more information. This option is only valid if a port descriptor is defined.

-S NAME
Use the given NAME for the status channel. The NAME must be a valid channel configured in the CCTK project configuration. This option may only be specified once. It is an optional parameter. If not specified, STDSTATUS will be used.

-d LEVEL
Sets the level of the debug output. Valid values are 1 through 5. Debug must be built into the application for this option to have any affect. Level 1 provides critical message, level 2 provides configuration information, level 3 provides additional debug on failure conditions, level 4 provides function entry and exit message, and level 5 provides detailed information on application flow. This is an optional parameter, if not specified, debug will not be displayed.

-h
Displays a terse list of the available options.

RETURN VALUES

PeerReceier will return SUCCESS if all goes well. It will return a positive value if an error occurs. 1 indicates an argument error, 2 indicates a CCTK error, 3 indicates a socket error, 4 indicates a system error, 5 indicates a error during execution, and 6 indicates an error during wrapup.

SEE ALSO

PeerSender

NAME

PeerSender -- Transmits CCTK data across a network

SYNOPSIS

```
PeerSender [-d LEVEL] [-i NAME] [-S NAME] [-k PATH] [-P NAME | ([-m MODE]
        [-l ADRESSS] [-L PORT] [-r ADDRESS] [-R PORT])]

    or

PeerSender –h
```

DESCRIPTION

PeerSender transmits CCTK packets across the network using either TCP or UDP packets. All types of CCTK packets can be transmitted from processed data packets to linked data packets to packet decom packets. In its most basic form, PeerSender only examines the packet header of the packet so any and all future packets will be supported as well. The receiver of data can either be the CCTK PeerReceiver process or another process written by a user. The protocol used by PeerSender is open and thus anyone can read data sent by the process. PeerReceiver is an application that allows the data sent by PeerSender to be read into a different CCTK system on a different host. The protocol is documented in the PeerProtocol.h header file. Also see the CCTK Administrators Guide for more information on configuring and using the PeerSender and PeerReceiver processes.

MODES OF OPERATION

PeerSender can operate in one of five different modes of operation:

- Registration
- Point-to-Point
- Connectionless
- Broadcast
- Multicast

In registration mode, PeerSender will wait for an initial connection from a PeerReceiver on the specified local port. When a connection is established, PeerSender will fork a new instance to handle the connection. The parent will return to waiting on the specified local port for further connections. The local port is a required parameter. The child will service the receiver. The receiver can "request" measurement data by sending registration request packets to the PeerSender. It is also possible to "unregister" for measurements as well as request a list of valid measurements. In this mode, only measurement and notice data can be passed from PeerSender to the receiver. In addition, the receiver must register for all information it wishes to receive.

In point-to-point mode, PeerSender will await for an initial connection from a receiver on the specified local port. When a connection is established, PeerSender will send all data received on its input channel to the PeerReceiver. The local port is a required parameter. If the receiver closes the port, then PeerSender will wait for another connection and repeat the cycle. In this mode, only a single receiver can receive the information.

In connectionless mode, PeerSender will send all data received on its input channel to a specified remote address/destination port using datagrams. The remote address and destination port are required. In this mode, only a single receiver can receive the information. Routers will typically forward UDP packets and thus this mode can be used to send data to hosts on a different network segment.

In broadcast mode, PeerSender will send all data received on its input channel to a specified remote address/destination port using datagrams. This mode differs from the connectionless mode in that the remote address is a broadcast address and the broadcast option for the socket is set. If no remote address is specified, "255.255.255.255" will be used by default. A remote port is required. In this mode, multiple receivers can receive the same information. Broadcast data is not passed through routers so the sender and receiver must be on the same network.

In multicast mode, PeerSender will send all data received on its input channel to a specified remote address/destination port using datagrams. This mode differs from the previous in that the remote address is a multicast address and the multicast options for the socket are set. A remote address and port are required. In this mode, multiple receivers can receive the same information. Routers can be configured to route multicast data and thus this mode can be used to send data to multiple hosts on different network segments.

The above modes of operation provides a wealth of options to the user. Both stream (TCP) and datagram (UDP) connection options are provide so that a user may choose the protocol that best suits their application. TCP has guaranteed reliability but increases overhead. UDP is not guaranteed but decreases overhead. On dedicated networks, in practice, UDP is as reliable as TCP.

The following table should aid in configuring PeerSender. It indicates which port and address arguments are required for the different modes of operation.

	Local Address	Local Port	Remote Address	Remote Port
Registration	optional	required	ignored	ignored
Point-to-Point	optional	required	ignored	ignored
Connectionless	optional	optional	required	required
Broadcast	optional	optional	optional	required
Multicast	optional	optional	required	required

CONFIGURATION

PeerSender can be configured through a port descriptor or on the command line options. If a port descriptor is specified (with the -P option), then it is not permissible to specify address information on the command (with the -l, -L, -r, or -R options).

The port descriptor is populated via the CCTK configuration database. Please see the peer-to-peer documentation for more information on configuring the PeerSender via the configuration database. The command line options are described below.

OPTIONS

-i NAME
NAME will specify the input channel name to use. The input channel will be used by PeerSender to receive both data and commands. It is a required parameter when operating in point-to-point, connectionless, broadcast, or multicast mode. It is an optional parameter when operating in registration mode. It may only be specified once. If it is not specified, it will default to NEXT_AVAIL_CHAN.

-k PATH
PATH will be used as the CCTK KPATH for this process. The specified path must contain a valid, running CCTK system. It is an optional requirement. If kpath is specified, the "KPATH" environment variable will be set to PATH. If kpath is not specified, kpath will be retrieved from the "KPATH" environment variable. If the environment variable is not present and kpath is not set on the command line, an error will be generated.

-l ADDRESS
Specifies the local address to use for the socket. ADDRESS should be of the standard form x.x.x.x or a hostname resolvable into an address using gethostbyname(). The local address is the address used to bind to on the sender host. This is optional for all modes. If not specified, any address will be used.

-L PORT
Specifies the local port to use for the socket. PORT should be a standard number or a service resolvable into a port using getservbyname(). This option is required for point-to-point and registration, optional for the other modes. If not specified, any available port on the system will be used.

-m MODE
MODE indicates the mode of operation for PeerSender. Valid modes of operation include: registration, point-to-point, connectionless, broadcast, and multicast. Each mode of operaton requires different port and address

information. The chart provided in the description above indicates which arguments are required for which mode. A further description of the modes of operation are provided above.

-P NAME
NAME will specify a port fd to use. The input channel of PeerSender will be registered in the specified port fd so that commands are routed correctly. It is an optional parameter. It may only be specified once. If no port is specified, then PeerSender will not be commandable via the CCTK system commands. If a port fd is specified, address and port information will be retrieved from the port descriptor and an error will be generated if address and port information is specified on the command line.

-r ADDRESS
Specifies the remote address to use for the socket. ADDRESS should be of the standard form x.x.x.x or a hostname resolvable into an address using gethostbyname(). This option is required for connectionless and multicast, it is optional for broadcast, and it is ignored by point-to-point and registration. For broadcast, it defaults to 255.255.255.255.

-R PORT
Specifies the remote port to use for the socket. PORT should be a standard number or a service resolvable into a port using getservbyname(). This option is required by connectionless, broadcast, and multicast. It is ignored by point-to-point and registration.

-S NAME
Use the given NAME for the status channel. The NAME must be a valid channel configured in the CCTK project configuration. This option may only be specified once. It is an optional parameter. If not specified, STDSTATUS will be used.

-d LEVEL
Sets the level of the debug output. Valid values are 1 through 5. Debug must be built into the application for this option to have any affect. Level 1 provides critical message, level 2 provides configuration information, level 3 provides additional debug on failure conditions, level 4 provides function entry and exit message, and level 5 provides detailed information on application flow. This is an optional parameter, if not specified, debug will not be displayed.

-h
Displays a terse list of the available options.

DIAGNOSTICS

PeerSender will return SUCCESS if all goes well. It will return a positive value if an error occurs. 1 indicates an argument error, 2 indicates a CCTK error, 3 indicates a socket error, 4 indicates a system error, 5 indicates a error during execution, and 6 indicates an error during wrapup.

SEE ALSO

PeerReceiver

NAME

PktDcom - The generic packet decommutation process

SYNOPSIS

PktDcom -i Input_Chan_Name -o Output_Chan_Name [-MkDh]

DESCRIPTION

The PktDcom process reads packets for decommutation from its input channel (Input_Chan_Name). The PktDcom process decommutates the packet data and sends linked data packets to its output channel (Output_Chan_Name) for further data processing (e.g. DataProc).

The input packets are single packet_dcom_pkt_t packets or CCTK superpackets (PACKET_DCOM_SUPER_PKT_TYPE) consisting of packet_dcom_pkt_t subpackets (PACKET_DCOM_PKT_TYPE).

The subpacket header contains a SID that references a packet decommutation descriptor. The subpacket data consists of raw data to be decommutated and flags that control the decommutation.

The subpacket header SID references a packet_dcom_des_t packet decommutation descriptor (PACKET_DCOM_DES_TYPE) in the Link Descriptor Table (LDT). The packet decommutation descriptor consists of link_SID_rec_t link SID records that define the decommutation parameters for the packet measurements.

The subpacket data decommutation flags control optional processing of the subpacket raw data. Flags exist to control raw dump processing as well as measurement list processing.

The subpacket data raw data consists of an array of bytes to be decommutated according to the link SID records in the packet decommutation descriptor.

OPTIONS

-i Input_Chan_Name

The Input_Chan_Name is a required text string command line argument. The Input Channel is the channel where PktDcom receives its input packets. The input packets contain raw data blocks to be decommutated.

-o Output_Chan_Name
The Output_Chan_Name is a required text string command line argument. The Output Channel is the channel where the raw measurement packets are distributed. The output packets contain individual raw decommutated measurements.

-M Max_Superpkt_Ratio
The Max_Superpkt_Ratio is an optional integer greater than zero that indicates the maximum number of input superpackets to process before an output superpacket should be sent. The Max_Superpkt_Ratio default value causes an output superpacket to wait until it is full before it is sent.

-k KPATH_Name
The KPATH_Name is an optional text string command line argument. The KPATH Name specifies the value to be used to set the KPATH environment variable. The KPATH environment variable is a required environment variable. The existing KPATH environment variable will be used as a default if one is not specified on the command line.

-D Dbg_Level
The -D option will enable debug mode. Debug mode causes verbose messages to be printed to a debug file in the log directory. The debug filename is DEBUG.PktDcom.### where ### is the PktDcom process ID. The Dbg_Level is an optional integer between 0 and 5 that determines how verbose the messages will be. Level 5 is most verbose.

-h

The -h option will print a help message to the screen. The help message indicates the current default values of the optional command line arguments.

RETURN VALUES

The PktDcom process returns SUCCESS_EXIT to the invocation environment if process initialization and termination was successful. One of the following values is returned to indicate the reason for failure if an initialization or termination error occurs:

```
OPT_ARG_INIT_EXIT_ERR     Error processing command line arguments.
CCTK_INIT_EXIT_ERR        Error establishing CCTK Kernel resources.
CONFIG_INIT_EXIT_ERR      Error configuring Unix resources.
STATUS_INIT_EXIT_ERR      Error updating process status.
PROC_STATE_EXIT_ERR       Error occurred after initialization.
CCTK_TERM_EXIT_ERR        Error terminating CCTK Kernel resources.
```

SEE ALSO

DataProc, PktDcomIfDefs.h

NAME

RetGraceGenPlot - Generates a grace plot from a retreival file.

SYNOPSIS

RetGraceGenPlot <retrieval_file>.delim

DESCRIPTION

RetGraceGenPlot is normally called by RetMon when a user requests a plot of retrieved data. It is possible, however, to call RetGraceGenPlot by hand on an existing file of retrieved data. RetGraceGenPlot MUST operate on the delimited version of the retrieved data. This is the retrieved data file with the .delim extension. Otherwise, the usage is simply:

```
RetGraceGenPlot <retrieval_file>.delim
```

After the data is processed by the script, grace will run and the plot will be visible.

SEE ALSO

RetGraceParseFd

NAME

RetGraceParseFd - Parses FD data from a delimited file and formats it for ingestion by the grace plotting tool.

SYNOPSIS

RetGraceParseFd <retreival_file>.delim <fd_name>

DESCRIPTION

RetGraceParseFd is used to correctly format measurement data contained within a retrieval file for ingestion by grace. RetGraceParseFd will generate a stream of data on standard out for the given fd_name that is formated correctly for grace to read in via a pipe. Currently, this script is used by RetGraceGenPlot, but it can be used by a user. The correct syntax of a grace command using this utility is:

```
xmgrace -source pipe -nxy "RetGraceParseFd <retreival_file.delim> <fd_name>"
```

Other options to control the format of the graph may be necessary. In addition, if the fd_name and/or file contains spaces, it may be necessary to correctly quote escape and quote the fd_name and/or file.

SEE ALSO

RetGraceGenPlot

NAME

Retriever - graphical front-end to CCTK retrievals

SYNOPSIS

Retriever -- [-k KPATH]

DESCRIPTION

Retriever is a graphical front-end to the CCTK retrieval. Retriever has the following features:

ability to save/load parameter files.
ability to specify measurements/commands to retrieve.
ability to specify time ranges to retrieve.
ability to specify measurement details.

OPTIONS

The only option that Retriever accepts at this time is -k. The -k option will specify the location of the CCTK kernel.

NAME

SetSystemState -- Set the CCTK project state.

SYNOPSIS

SetSystemState [-m mode_info] [-t test_description] [-k kpath] [-u user_notes] <-s new_state>

DESCRIPTION

SetSystemState is a simple application used to set the current state and associated state information of a CCTK project. Options are available to set the project description (-t), the user notes (-u), and mode information (-m). The new state is given using the -s option. The new state must be one of the valid CCTK states (DOWN, STARTING_UP, UP, SHUTTING_DOWN, DOWN, FORCED_DOWN. Typically, SetSystemState will be used to force a test down after a system failure.

The SetSystemState application must know the location of the system_state.xml file. The following rules determine where SetSystemState will find the system_state.xml file:

- If -k is specified, the application will look for "system_state.xml" in that directory.
- If -k is not specified, the application use the contents of the KPATH environment variable as the directory where "system_state.xml" should exist.
- If KPATH is not set, then the application will search the current directory. If a "system_state.xml" file exists there, it will be used, but a warning will be displayed.
- If all of the above fail, an error message will be generated.

OPTIONS

-h
Displays the short help on the use of SystemState.

-m mode_info
Sets the mode information in the system_state.xml file.

-t test_description
Sets the project description in the system_state.xml file.

-k kpath
Search for "system_state.xml" in the given directory.

-u user_notes
Sets the user notes in the system_state.xml file.

-s new_state
Sets the system state to the given state.

SEE ALSO

ClearSystemStateOnBoot, SystemState

NAME

SimGui - graphical user interface to the simulation scripting language

SYNOPSIS

SimGui -- -version SimGui [-k kpath] [-c command file ...] [-m measurement file ...] <script file>

DESCRIPTION

SimGui is a graphical user interface to the simulation scripting language. It provides an easy-to-use interface into many of the features of the simulation scripting language for users not wishing to learn the details of the language themselves.

If the -k option or if the KPATH environment variable is set, the SimGui will attempt to load command and measurement data from the specified kpath directory.

Note:

command and measurement data files are cumulative, all available command and measurement data files will be loaded if possible, this includes those in the kpath and any specified on the command line. Command and measurement data files are the GUIs link to CCTK. They are used only for user reference and are not required to run the simulation GUI.

For more information on the usage of SimGui, please see the on-line help.

OPTIONS

--

This is a required command line argument to indicate the start of local arguments. If this is ommitted, the -c option will be interpreted by the Tcl interpreter and an error will be produced.

script file
The user can specify a simulation file to load on the command line.

-k kpath
If a kpath is specified, the simulation GUI will attempt to load command and measurement information from that kpath.

-c command file
The user can specify a file that contains the list of valid command FD's in the system that is being simulated. This list is then used to prompt the user for valid commands. This argument is optional.

-m measurement file
The user can specify a file that contains the list of valid measurement FD's along with description and limits for the system that is being simulated. This list is then used to prompt the user for valid measurements and provide information on those measurements. This argument is optional.

–version
This option will display version and copyright information

SEE ALSO

SimEngine, Simulation Tutorial, Simulation Language Reference

NAME

SimpleHist - perform a simple historical retrieval

SYNOPSIS

SimpleHist [-rpm] [-l path] [-f filebase] [-b start_time] [-e end_time] -o output_file fd [fd ...]

DESCRIPTION

SimpleHist will perform a simple historic retrieval based upon a set of command line arguments. It gives developers and users a simple, fast, convient way to perform retrievals. It does not provide an interface to all of the features of retrieval, for more complex retrievals, use parameter files.

NOTE: the start time and stop time specified to SimpleHist is of a different format than that used by HistRetrieve. HistRetrieve uses the archive "restrictive" format of "YYYY/DDD/HH:MM:SS.MMM". Any time format other than that passed to HistRetrieve will fail. Since SimpleHist(), is, simplier. It uses a simplified time entry scheme. Any time valid for the Tcl "clock scan" command will work. Therefore, for most normal uses, you only need to specify HH:MM:SS. SimpleHist() will automatically convert to the more restrictive format for you.

OPTIONS

-r

retrieve raw data (can be used with -p).

-p

retrieve processed data (can be used with -r).

-m

retrieve system messages.

-b

start time (optional, will default to beginning of archive).

-e

stop time (optional, will default to end of archive).

-l

path (optional, will use current directory if not specified).

-f

filebase (optional, will use TAM if not specified).

-o

output file name.

fd

a list of fds to retrieve, all data for a particular FD will be retrieved. Both commands and measurements can be listed here.

If no options are specified, the default is to do a processed retrieval for the entire archive. At least one FD should be specified and an output file.

NAME

SimpleNrt - perform a simple near real-time retrieval

SYNOPSIS

SimpleNrt [-rpm] [-b start_time] [-e end_time] -o output_file fd [fd ...]

DESCRIPTION

SimpleNrt will perform a simple near real-time retrieval based upon a set of command line arguments. It gives developers and users a simple, fast, convient way to perform retrievals. It does not provide an interface to all of the features of retrieval, for more complex retrievals, use parameter files.

OPTIONS

-r

retrieve raw data (can be used with -p).

-p

retrieve processed data (can be used with -r).

-m

retrieve system messages.

-b

start time (optional, will default to beginning of archive).

-e

stop time (optional, will default to end of archive).

-o

output file name.

fd

a list of fds to retrieve, all data for a particular FD will be retrieved. Both commands and measurements can be listed here.

If no options are specified, the default is to do a processed retrieval for the entire archive. At least one FD should be specified.

NAME

StatMon - monitor kernel process status

SYNOPSIS

StatMon [kpath]

DESCRIPTION

StatMon is an interactive display of the current status of the kernel processes. StatMon must have a kpath to assist in accessing the status shared memory segment. The kpath may be specified on the command line or by an environment variable named KPATH. StatMon has a help screen which lists the options the user may enter to control the StatMon display. Hit the "h" key to display the help screen.

RETURN VALUES

Upon normal termination, StatMon returns zero. Otherwise, an error message is output and one is returned.

NAME

SysMsgGui

SYNOPSIS

SysMsgGui -- [-l] [-e] [-d time|criticality] [-k kpath] [-f message_file] [-m message_list]

DESCRIPTION

SysMsgGui is the System Message Graphical User Interface for the CCTK system. It provides a flexible, convenient way for a "user" to view and examine incoming system messages.

See the user documentation for the use of the system message GUI.

OPTIONS

-l

This option will cause the system message GUI to load the entire contents of the message file rather than seeking the end of the file. If the system has been running for a long time or has experienced a lot of errors, this could possibly dump thousands of errors into the system message GUI adversely affecting performance.

-e

If this option is enabled, the GUI will disable all "exit" capabilities via the GUI. This will not disable any of the window manager capabilities, but should prevent a naive user from "exiting" the application.

-d time|criticality

This controls the display mode in which the system message GUI will initialize. If t or time is specified, the default display is a single time sorted pane (this is the default). If c or criticality is specified, the default is a three paned window sorted by criticality.

-k kpath

The user can optionally specify the KPATH. If this is not specified, the KPATH will be retrieved from the environment.

-f message_file

The file that contains the messages. SysMsgGui will watch this file. As messages are added to the file, SysMsgGui will read messages from the file and display them to the screen. If not specified, KPATH/log/system_messages.d is used.

-m message_list

The file that contains the list of "valid" messages. This is used as an aid to the user when building filters. It is necessary to allow a user to correctly create filters. If not specified, KPATH/msglist is used.

NAME

SystemState – Query the current state of a CCTK test

SYNOPSIS

SystemState [-k kpath] [-s] [-c] [-i] [-p num|-d]
or
SystemState -h

DESCRIPTION

SystemState is a simple application used to query the current state of a CCTK test and modify the state history of the test. There are three categories of information that can be returned by SystemState. The -s flag returns information about the current state. The -c returns information about the current configuration. The -i flag returns information about the state history. Multiple flags can be specified. Data will always be return in the following order: current state, current configuration, and state history, regardless of the order of the command line flags.

SystemState can modify the state history of the test through the -p and -d flags. The -p and -d flags are mutually exclusive. Only one may be specified on the command line. -p is used to prune the history. -d is used to delete the history list entirely. If -p is specified, an argument is required that indicates the number of elements that will remain after pruning.

The SystemState application must know the location of the system_state.xml file. The following rules determine where SystemState will find the system_state.xml file:

If -k is specified, the application will look for "system_state.xml" in that directory.
If -k is not specified, the application use the contents of the KPATH environment variable as the directory where "system_state.xml" should exist.
If KPATH is not set, then the application will search the current directory. If a "system_state.xml" file exists there, it will be used, but a warning will be displayed.
If all of the above fail, an error message will be generated.

OPTIONS

-h

Displays the short help on the use of SystemState.

-s

Display the current state of the test.

-c

Display the current configuration of the test.

-i

Display the state history of the test.

-k kpath

Search for "system_state.xml" in the given directory.

-p num

Prune the state history, leaving num elements remaining.

-d

Remove the entire state history list.

NAME

TamArs - Temporary Archive Media Archival and Retrieval Services

SYNOPSIS

TamArs [-a archive_channel] [-f ars_config_file] [-k kpath]

DESCRIPTION

TamArs provides archival and retrieval capabilities for real-time data processed by the CCTK system. Data to be archived is sent to this process via the archive_channel. The data packets received are bundled into archive buffers and written to the archive media. Data which has been recorded to the archive media may be retrieved and formatted into textual reports, or may be graphically displayed using COTS products.

This process writes the archived data to a temporary archive on a local hard disk. This process also passes the archive buffers to the Pam Archive process, so that the data may be saved on permanent removable media (such as an optical disk or DAT tape). The temporary archive is used for near real-time retrievals. If the temporary archive becomes full, the oldest data in the archive is overwritten by the new data (i.e., the temporary archive acts like a circular buffer). Data written to the temporary archive can be retrieved until the data is overwritten (by the archive "wrapping" when it becomes full) or until a new temporary archive is created. Currently, a new temporary archive is created each time the TamArs process is initialized/started. In future EVOs, a new temporary archive will be only created when a test is run with a different data base than the previous test. Data from previous tests and those using a different data base can be retrieved from the permanent archive media. In general, retrievals from the temporary archive media provide faster access to the archived data than retrievals from the permanent archive. If a disk error occurs writing to the temporary archive, recording to the temporary archive stops; however, the data already written to the archive can still be retrieved.

TamArs interfaces with the Archive Control process via shared memory for requests to start and stop the archival activity. It notifies the Pam Archive process of data to be archived via a channel. It also interacts with the Retrieval GUI via shared memory, for requests to initiate, pause, resume, and terminate retrievals.

TamArs tracks updates to FD's, so that the correct FD data is output by Retrievals for FD's updated after the archive processing begins.

OPTIONS

-a archive_channel

This parameter is optional. Data to be archived by TamArs is sent via this channel. If this parameter is not specified, it defaults to STDARCH.

-f ars_config_file

This parameter is optional. If present, it specifies a disk file containing parameters which modify (override) default values used by TamArs. See ars_config(9D) for a list of the default values which may be overridden.

-k kpath

This parameter is optional. The TamArs process must have a kpath to assist in accessing the channels and shared memory tables. If not specified on the command line, the value is obtained from an environment variable named KPATH.

RETURN VALUES

If an error occurs during initialization, an error message is output and TamArs exits. Otherwise, TamArs runs until a terminate signal is received.

SEE ALSO

ArchiveControl, PamArchive, ars_config, Retrieval

NAME

ProjectManager - Manages a CCTK project

SYNOPSIS

ProjectManager [-f configuration_file] [-k kpath] [-n user_notes] [-i|-v|-q] [-u mode|-d]

DESCRIPTION

ProjectManager performs activities associated with managing a CCTK project. These activities include:

Creating the needed kernel resources.
StartUp of core kernel processes.
Monitor of core kernel processes.
Monitor of user initiated processes.
ShutDown of the project.
Updating the system_state.xml file.

More details on each of the above tasks is provided below.

ProjectManager will behave differently depending on how it is executed. These behaviors include:

StartUp or ProjectManager -u
With this behavior, ProjectManager will start the project in the given mode. If no mode is given, the default mode listed in the configuration file will be used. After the system has been started successfully, ProjectManager will continue on with its monitor and shutdown tasks.
ShutDown or ProjectManager -d
With this behavior, ProjectManager will assume that a project is running in the given kernel path directory and will attempt to terminate the project. It will clean up all resources found. It will kill any process associated with the project that is still running.

ProjectManager will read in a configuration file to determine how to configure CCTK. This configuration file defaults to the name project_config.xml. The default name can be overridden on the command line by using the -f option. See the associated documentation for a detailed description of this file.

ProjectManager also needs to identify a KPATH where important project information will be stored. The kernel path is typically defined by the environment variable KPATH, however, it can be defined in several other ways. The following describes the order of precedence used by ProjectManager to identify a KPATH. The first item found in this list will be used.

The command line option, -k.
The environment variable, KPATH.
The kernel path specified in the configuration file.
The current directory.

STARTUP DETAILS

When ProjectManager is tasked with starting the system, it will perform the following steps:

Verify there is no project operating in the current directory.
Create a new UNIX session.
Change the state of the system to STARTING_UP.
Find and parse the project_config.xml configuration file.
Create environment variables specified in the configuration file.
Create the CCTK status block.
Create shared memory specified in the configuration file.
Create message queues specified in the configuration file.
Create channels specified in the configuration file.
Execute the processes listed in the configuration file.
Verify that all processes transition correctly to the running state.
Detach from the controlling terminal and move to the background.
Change the state of the system to UP.

If any of the above tasks fail, ProjectManager will "undo" the steps it has completed thus far and then change the state of the system to STARTUP_FAILED.

ProjectManager will track the pids of all critical processes it starts. It stores this list of pids for later use during monitoring activities.

MONITORING DETAILS

After completing the startup tasks, ProjectManager will monitor the running CCTK system. ProjectManager will perform the following monitor tasks, once per update cycle:

Call waitpid() to see if any of the critical processes have exited.
Evaluate the operational state of all running processes. Flag any non-responsive process as TARDY or AMUCK.

SHUTDOWN DETAILS

ProjectManager will shutdown a CCTK project when it receives the SIG_KTERM signal. At the time of shutdown, the following tasks will be performed:

Change the state of the system to SHUTTING_DOWN.
The system status block will be updated to indicate that the system is shutting down.
Using the waitpid() call, allow all child processes to exit. Continue until all child processes exit or a given time period passes without any child exiting.
The PROC directory will be searched. For each entry remaining in the PROC directory, the following algorithm will be used:

```
        If the process is still alive (determined with kill -0):
            If the process was started by this ProjectManager:
                Send a TERM signal to the process.
                If the process is still alive (determined with kill -0):
                    Send a KILL signal to the process.
                EndIf
                Remove the PROC directory entry if it still exists.
            EndIf
        Else
            Remove the PROC directory entry.
        EndIf
```

Remove the configured channels.
Remove all remaining message queues.
Remove all remaining shared memory segments.
Remove the CCTK status block.
Change the state of the system to DOWN.

DISPLAY OUTPUT

ProjectManager will operate in one of three output modes. They are quiet (-q), verbose (-v), and interactive (-i). Quiet will generate no output except for critical errors. Interactive will generate output in a user-friendly format. Interactive should be used when a user manually executes the ProjectManager command. Verbose is used to provide detailed information in a easily-parsed format to an application that is communicating with ProjectManager via a pipe. Typically, if ProjectManager is started by another application, say SystemControl, the verbose option will be used. If ProjectManager is started at the shell, then interactive will be used.

ProjectManager will use stty() to check and see if it is attached to a terminal. If it is, it will default to interactive. Otherwise, it will default to verbose.

Regardless of the mode, once the system is placed in the running state, ProjectManager will fork and place itself in the background, returning control to the calling process. This is done so that if the controlling process aborts, the CCTK project will not be taken down as well.

OPTIONS

-d

Indicates the ProjectManager should be run in "ShutDown" mode and that the project resources associated with the current KPATH will be cleaned up.

-f configuration_file

Specifies a configuration file to use. This overrides the default configuration file of project_config.xml.

-h

Lists the available options for ProjectManager.

-i

ProjectManager will operate in user interactive mode. This assumes that a user initiated the command from the command prompt. If stty() returns true, this is the default mode. Details on the startup will be displayed in a user readable format.

-k kpath

Specifies a kpath to use which will override all other possible kpaths.

-n user_notes

Specifies "user_notes" to place inside the system_state.xml file. If this option is not specified, then no user notes will be placed in system_state.xml.

-q

ProjectManager will operate in quiet mode. Only critical errors will be reported.

-u mode

Indicates that ProjectManager should be run in "StartUp" mode and that the CCTK project should be started. The mode is optional. This indicates which mode to select from the configuration file. If no mode is given, the default listed in the configuration file will be used.

-v

ProjectManager will operate in verbose mode. This assumes that another process is communicating with ProjectManager via a pipe. Details on the startup will be displayed in an easy to parse machine format.

NAME

csim - CCTK version of the simulator

SYNOPSIS

csim [-e|s] <-o output_channel> [-i input_channel] [-p port_fd [-p port_fd] ...] [-k kpath] [script_file ...] csim -h

DESCRIPTION

csim is the command used to run the CCTK version of the simulator. The CCTK simulator will accept all the core modules and produce output in the form of CCTK linked data packets.

The output_channel is a required parameter. This indicates the name of the output_channel to which all linked data packets will be written.

The input_channel is optional. An unlimited number of input channels may be specified. The simulator will attempt to read commands from each input channel once during each of its processing cycles.

csim has command capabilities. If a command is received, the appropriate command module will be messaged through the IssueCommand() method. csim will register for commands in the port descriptor for each port fd specified on the command line. Multiple ports can be specified. The registered input channel will always be a next available channel.

The user can optional specify the kpath which indicates where the CCTK configuration is found. If kpath is not specified, the process will query the KPATH environment variable.

csim also has the capability of using either an internal timer that is managed by this instance of the simulator or an external timer that is managed as part of CCTK. With no arguments, the internal timer is used. If the -e argument is specified, the external timer is used, as long as no errors occur. Or, if the -s argument is specified then the simulator will be capable of simulating and propogating time to CCTK as well.

Finally, the user can specify a script file. If a script file is specified, the simulation commands in that script file are processed. If no script file is specified, then csim will enter an interactive mode and accept input from the user via a command line.

If -? or -h is specified, help information will be displayed.

NAME

dsim - Debug version of the simulator

SYNOPSIS

dsim [-o output file] [script file]

DESCRIPTION

dsim is the command used to run the debug version of the simulator. The debug simulator will accept all the core commands and produce output to either standard out or a file. If a file is specified on the command line, all output will be redirected to that file. If no file is specified, output will be directed to standard out.

dsim will display a single line each time the update cycle starts and/or ends. An additional message will be displayed for each measurement that changes.

dsim has no command capabilities and simply displays a message when the CheckForCommands() method is messaged.

If dsim is called with a script file specified, dsim will process all commands contained within the script file. If no file is specified, dsim will enter an interactive mode and accept input from the user via the command line.

NAME

gsim – Glg version of the simulator

SYNOPSIS

gsim -s <server name> [script file]

DESCRIPTION

gsim is the command used to run the glg version of the simulator. The glg simulator will accept all the core commands. Changes to the simulated measurements will cause the appropriate resources to update at the glg server specified by <server name>.

gsim has no command capabilities.

If gsim is called with a script file specified, gsim will process all commands contained within the script file. If no file is specified, gsim will enter an interactive mode and accept input from the user via the command line.

GLOSSARY

The following terms and abbreviations are used throughout the CCTK documentation set:

Actor – Command and measurement models used in CCTK simulation.

Actor groups – Collections of actors.

Analog measurement – Measurement data that is represented as a continuously variable quantity.

ANSI – American National Standards Institute. See http://www.ansi.org/

API – See Application Programming Interface.

Application programming interface – A library of functions and associated data structures that allows application software to link with CCTK.

ARS – Archive and Retrieval Subsystem.

BDT – Bus Descriptor Table.

CCB – See Channel control block.

CCT – Command and Control Technologies Corporation.

CCT_HOME – Environment variable that specifies the installation directory of CCTK.

CCTK – Command and Control Toolkit.™

CCTK Client – The general reference to the program *CctkClient*, which is the main user interface for operating CCTK.

CCTKPROJECTLABEL – An environment variable that specifies an alternate term for a CCTK project.

CDB – See Configuration Database.

CDT – See Command Descriptor Table.

Channel – A channel is a means of interprocess communications. There are two types of channels, private and public. A private channel is an exclusive point-to-point connection between two processes. Private channels do not go through a distributor. A public channel is one to which any process can write.

Channel control block – An area of memory contained within the channel shared memory block that stores state information related to the channels configured in a particular instance of CCTK.

Client – Any process that requires services from another process within CCTK.

Command – A command represents a control input to an end item or operational system. The characteristics of a command are defined in command descriptions.

Command Descriptor Table – Contains information, including name and attributes, associated with system commands.

Command Response – The data sent in response to a command.

Condition Notice – A condition notice is the set of static attributes of a system message defined in the configuration database. Condition notices do not "occur." There is no temporal aspect associated with a condition notice and they are not passed around the system. There are a set number of condition notices defined for each project. A condition notice is analogous to a "class" and a system message is analogous to an "object" of that class. When a system message is generated, a instance of a condition notice is instantiated in the system.

Configuration Database – Persistent configuration repository for all measurement parameters, command parameters, and interface specifications within CCTK.

Configuration Descriptor – A configuration record or object.

Configuration Item – A configuration item is a hardware or software entity, or an aggregation of both, which is designated for configuration management. Each configuration item has a unique set of function and/or physical attributes.

DAC – Data acquisition; see PCM data acquisition application.

Data Compression – Filtering of data to reduce bandwidth requirements.

Data Conversion – Linearization and end conversion of data.

Data Retrieval Request – Request for near real-time or permanent archived data.

Descriptor – A single named object within CCTK.

Discrete Measurement – Measurement data that is represented as one of two discrete states, such as ON or OFF.

Display Monitoring – Display monitoring is a mechanism by which a user can select and view the current value(s) of one or more measurements.

Distributor – A process that routes data through the Kernel and to external interfaces.

DMON – See Display Monitoring.

DTD – Document Type Definition, which is an XML construct that defines a document structure with a list of XML elements. See http://www.xml101.com/ for more information.

DTDPATH – An environment variable that specifies the location of CCTK DTD files.

Dynamic Displays – Dynamic Displays associate attributes (e.g. color, blink, position, etc.) with screen objects that can, in turn, be associated with dynamic values (e.g. measurement values).

EIA – Electronic Industries Alliance.

End Item – Application device or system under the control of CCTK. The end item is a source of raw measurement data, recipient of commands, and generator of command responses.

EOF – End of file.

Exception Condition – A condition created when a measurement satisfies an event against boundary values, expected values, and masks.

Exception Event – The transition between states (positive and negative) that occurs when an expression is evaluated.

Exception Expression – An algorithm that determines the states associated with measurement values. An expression may be atomic or compound formulas consisting of operators and conjunctions.

Exception Monitoring – Generic, system provided application for monitoring exception conditions. Exception monitoring presents exception messages to the user upon detection of exception conditions detailing the conditions of the exception.

Exception Notice – An asynchronous notification of an event provided to an application.

Exception State – Persistent representation of the last exception event associated with a single measurement.

FD – See Function Designator.

Fieldbus – An open standard for industrial control I/O buses defined by ANSI/ISA-50.02, Part 2-1992.

FIFO – First in, first out.

FFI – See Frame Format Identifier.

Frame Format Identifier – An identification number within a PCM frame used to uniquely identify the contents of a frame.

Front End – Communication interfaces that connect end items to CCTK.

FSP – Frame Sync Pattern.

Function Designator – A unique symbolic name representing related data within CCTK. Function designators are defined by CCTK users and are usually associated with sensors and effectors whose value is resolved by a CCTK external interface or system application during project operations. The FD is the principal handle for data and command access within CCTK. FD's are also associated with CCTK information such as condition notices and processing chains.

GLG – Genlogic; GLG is the graphic builder and display creation tool from Genlogic Corp.

GMT – Greenwich Mean Time.

GPS – Global Positioning System.

Graphical User Interface – The interface through which clients access system functionality. This interface includes software elements (i.e. displays, controls) as well as hardware elements (i.e. workstations, input devices). Graphical and textual screen formats used to display data and accept client input.

GUI – See Graphical User Interface.

HCI – Human-computer interface; see Graphical User Interface.

Health and Status Data – Information about CCTK kernel processes that indicate of the state of the system.

HTTP – Hypertext transfer protocol.

Human-Computer Interface – See Graphical User Interface.

IBS – Interbus; IBS is a SCADA bus protocol from Phoenix Contact Corp.

Interlock Processing – The processing of a set of conditions upon which command issuance is contingent. Also referred to as "Prerequisite Control Logic."

IP – Internet protocol.

IPC – Inter-process Communication. See UNIX system call documentation.

IRIX – The UNIX-based operating system produced by Silicon Graphics.

ISA – Instrumentation, Systems and Automation Society. See http://www.isa.org/.

Kernel – The core real-time services of CCTK. The kernel provides services to applications and end item interfaces for data processing, data and command distribution, and time management.

Kernel Programming Interface – An application programming interface consisting of functions provided directly by the CCTK kernel.

KMSG – Directory containing message queue information.

KPATH – An environment variable that points to the current project directory. *KPATH* is used by every CCTK process to locate the resources associated with a particular project. *KPATH* typically needs to be set if the user is going to be performing operations from the UNIX command line.

KPI – See Kernel programming interface.

KSHM – An environment variable that indicates the directory containing information on the shared memory segments created for a CCTK project. The name of the each file in the *KSHM* directory equates to the name of the shared memory segment. The contents of the file contains the shared memory key associated with the segment. Using this key, it is possible to associate the shared memory segments listed by the UNIX *ipcs* command with those used by a CCTK project.

LDT – See Link Descriptor Table.

Link Descriptor Table – An internal CCTK table that contains information associated with external commands and link records which link a measurement with an external interface.

Linked Data – A single raw measurement sample that has been acquired, time stamped, and associated with its FD handle, but not yet processed.

LSB – Least significant bit.

MDI – Multiple document interface.

Measurement – A measurement is a unit of information that is received from an end item or application. The characteristics of a measurement are defined in the CCTK configuration database. Measurements can be represented by an analog, discrete, integer, or byte array value.

Minor Frame – A minor frame is a constant sized sequence of data including a synchronization pattern for hardware identification and optionally a frame ID for software identification to uniquely identify the frame. A major frame consists of one or more minor frames.

MSB – Most significant bit.

NACK – Negative acknowledge.

NDT – Notice Descriptor Table.

NRZ – Non-return to zero.

NTP – Network Time Protocol.

Operator – A person concerned with the operation and/or administration of a control system associated with CCTK; often used interchangeably with "user."

Packet – Any set of data associated with a packet header allowing it to be communicated through the Kernel. A packet may be the transport structure for a processed data measurement, a message, a command, a configuration descriptor, or any other type of data that circulates through the system.

PAM – Permanent Archive Media.

PCI – Personal Computer Interface. The electronic interface between Intel-class processors and peripherals.

PCM – Pulse code modulation.

PCM Data Acquisition Application (DAC) – The CCTK software program that processes PCM telemetry data.

PCMPATH – An environment variable that indicates the location of the PCM Telemetry Interface software. The PCMPATH environment variable is used by the installation script and the user environment scripts.

PDI – Payload data interleaver (a data format associated with the Space Shuttle).

Permanent Archive Media – Long-term storage for recorded data. Each PAM has a type of media and a volume. A volume is a single instance of a type of media. It may also be referred to as historical archive media. The media may be any standard storage device format, e.g., DAT or CD.

Point-to-Point Channel – A channel that is directly connected to two processes without the need for a distributor for routing the data.

POSIX – Portable operating standard for Unix. See *POSIX.4 Programming for the Real World*, O'Reilly & Associates, Sebastopol, CA, 1995.

Pseudo Function Designator – A special type of Function Designator that can be used for storage of data and to aid in communication. Typically used to designate derived data.

Qt – A Trolltech product that supports portable user interface applications; see http://doc.trolltech.com/.

Raw Frame – A block of data that is acquired from the end item.

Raw Measurements – Data that has not been linearized or scaled (counts).

Real-Time Control Server – The computer that hosts the CCTK server software.

Recorded Data – All measurement, command, event, message and other data that are recorded on temporary and permanent archive media for retrieval and Analysis. This includes End item Data and Configuration data.

RT – Real time.

RTCS – See Real-time control server.

RTTM – Real-time table manager.

SCADA – Supervisory control and data acquisition.

SFID – See Sub-frame identification.

SGI – Silicon Graphics, Inc.

SID – See Software Identifier.

SIGCHANSIGNAL – The CCTK signal used to indicate a packet has been placed on a channel when operating a channel asynchronously.

SIGCHANWAKEUP – An internal channel notification signal.

Simulated Universal Time – A simulated clock used by the CCTK simulation function.

Software Identifier (SID) – Internal CCTK index key for access to command measurement information. The SID is generated at configuration build time. There is one unique SID for every Function Designator (FD).

SQL – Standard Query Language.

Stale Data – Data that has not changed in value for a minimum of a user specified number of data samples.

STDARCH – The standard CCTK channel for information archival. All data sent to the STDARCH channel will be archived by the ARS.

STDDIST – Legacy CCTK channel that is no longer used.

STDMSG – The standard CCTK channel for system messages. All messages sent to the STDMSG channel will be processed and forwarded to interested processes, the system message file, and the archive subsystem.

STDRESP – Standard CCTK channel for command responses.

STDSTATUS – The standard CCTK channel for status information.

Sub-frame identification – A cyclic sequence number contained within a telemetry minor frame used by the decommutator hardware and/or software to verify major frame lock.

Super packet – A packet which contains multiple subpackets. Superpackets are used to increase efficiency by reducing overhead when transmitting data.

SUTC – See Simulated Universal Time.

System Message – An asynchronous event that notifies the user some action has occurred. System messages have a set of static attributes defined in the configuration database and a set of dynamic attributes generated at instantiation.

TAM – See Temporary archive media.

Temporary Archive – An archive that stores data in the local file system before being written to permanent archive.

Temporary Archive Media – Archive media represented by a standard UNIX file where data is stored before being written to permanent media.

TCL – Tool Command Language, a freely available platform-independent, string-based, interpreted command language. See http://www.neosoft.com/tcl and http://hegel.ittc.ukans.edu/topics/tcltk/index.html for more information.

Telemetry Data Stream – A continuous serial bit stream of time division multiplexed data.

TIX – Tk interface extension. Tix is an open open source extended widget set for Tcl/Tk. See http://tix.mne.com/ and http://tix.sourceforge.net/ more information.

UTC – Universal Time, Coordinated. See http://www.time.gov/timezone.cgi?UTC/s/0/java.

Valid – Raw and processed data values lie within a defined, valid range.

X Protocol – X Window System technology provides display and management of graphical information for UNIX-based computers. The underlying protocol is defined by X.org; see http://www.x.org.

XCDB – XML Configuration Database – See Configuration Database.

XIO – Extended input/out.

XML – Extensible Markup Language; a self-describing markup language defined by the World Wide Web Consortium designed to describe data. XML is an open technology; see http://www.w3.org/XML/.

INDEX